Synthesis of Dataflow Graphs for Reconfigurable Systems using Temporal Partitioning and Temporal Placement

Dissertation

A thesis submitted to the
Faculty of Computer Science, Electrical Engineering and Mathematics

of the

University of Paderborn
in partial fulfillment of the requirements for the degree of *Dr. rer. nat.*

by

Christophe Bobda

Paderborn

Acknowledgments

This work was carried out at the Department of Computer Science, Electrical Engineering and Mathematic of the University of Paderborn from 1999 to 2003.
Firstly, I am thankful to my supervisor, Professor Rammig. I consider it a great accomplishment to have completed this dissertation under his direction. I was very fortunate to work as member of his chair. Therefore, i could obtain all the necessary resources i needed for my work. Prof. Rammig has always been ready to help me and he gave me the best advice all the time. No advisor would have been better than him. I am grateful to the staff at the working group "Design of Parallel Systems" for their valuable help during my work.
I want to express my gratitude to the official examiners of this thesis, Prof. Franz J. Rammig, Prof. Wolfram Hardt, Prof. Jürgen Becker, Prof. Wilfried Hauenschild and Dr. Peter Pfaler. Their constructive criticism improved the quality of the thesis considerably.
I am particulary thankful to my collegue Thomas Lehmann, who introduced me to the topic of Hardware Design and influence my work with his strong engineering way of thinking. Thank you Tom.
In the same way I want to thank my collegues Achim Rettberg, Stefan Ihmor, Klaus Danne, Mauro Zanella and Thomas Lehmann for their strong cooperation in differents projects. Their advice help me in different way during my work. Thanks also to my working students Nils Steenbock and André Linarth who implemented a considerable part of my concepts.
I am also grateful to Romain Bertrand Kamdem, Achim Rettberg, Klaus Danne who spend a great part of their valuable time to review my work and provide me the necessary advice to improve the quality of my thesis. Thanks also to Gregorio Reid who spent a lot of time to revise the language of the manuscript.
I would also like to thank my brothers and sisters Samuel Njoko, Hervé and Severin Kouam, Francis and Michele Kouayep, Jonas Kiegain and Augustin Kamnaing, my friends Norne Behn, Dr. Etinne Nitidem, Alain Fabo, Pascal Djiadeu, Hugues Njiende, Guy Douale, Ernest Manfor and Thomas Kameni for the moral support I got from them during the time I spent on this work.
Finally, I want to aknowledge that without the support of my loving wife Huguette and my son Jan-Wilfried, this thesis could not even have been attempted. Their patience and support throughout these long years has been crucial. Because I spent far less time at home than I wanted, Huguette ran the household as I pursued my scientific activities. The care of our son, Jan-Wilfried and our comming baby, has been in Huguette's capable hands. Thank you Huguette!

Paderborn, Mai 2003

Abstract

We provide in this thesis our contribution in the area of reconfigurable system synthesis. We consider reconfigurable systems constructed from one or more general purpose processors (GPP) and a set of reconfigurable processing units (RPU). Given an application to be implemented on this architecture, a hardware/software partitioning step is used to differentiate between the part of the application to be executed on the hardware and the part to be executed in software. The synthesis of a reconfigurable system consists of the hardware/software partitioning process as well as the implementation of the software part on the GPP and the hardware part on the RPU. The part to be implemented on the RPU is provided as a dataflow graph (DFG). This thesis deals with the part of the application to be implemented on the RPUs. The RPUs targeted are field programmable gate arrays (FPGAs). Because the DFGs to be implemented in the FPGAs are usually too large to fit in a single FPGA, they must be partitioned in to blocks. These blocks are then successively downloaded in to the FPGA to compute the desired function. If the FPGA can not be partially reconfigured, then the blocks are used to configure the entire device. In this case, the partitioning process is called **temporal partitioning** otherwise it is a **temporal placement**. Our contribution consists of the development of various algorithms to solve the temporal partitioning and the temporal placement problems.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

General purpose processors offer the possibility to implement all kinds of functions using the same device. Although highly flexible, general purpose computing relies in general on very high clock rates for performance. Instruction execution incurs significant overhead because of the reliance on the Von Neumann paradigm. First, an instruction is fetched and decoded, then the operands are read and the operation coded in the instruction to be executed. Finally, the result is stored in memory.
When a dataflow oriented function is expected to be too slow on a general purpose processor, it is time to consider a hardware implementation. This is usually done in an ASIC (Application Specific Integrated Circuit) in which the function is hardwired once and cannot be changed again. Hardware implementation on ASICs is usually efficient. First, because the overhead caused by the instruction fetching and decoding, the reading and storage of data is removed. Second, because the ASIC is optimized for only one function. ASICs are, in turn, not flexible. Functions implemented in ASIC devices will remain unchanged for the life of the device. Moreover implementing a function in ASIC is a long and difficult process which incurs high NRE (non recurring engineering) cost. This cost can only be amortized for a very high volume production. Ideally, we would like to have the flexibility of the GPP and the efficiency of the ASIC in one device, which can be used to implement different functions independently of the volume required to amortize the NRE cost.
Combining the flexibility of the GPP and the efficiency of ASIC in one device has been proven to be a good solution. A new class of processing unit called a reconfigurable processing unit (RPU) has been created and their performance has been shown to be far greater than those of the GPP. RPUs have opened up new possibilities in ASIC emulation by reducing the NRE cost. It has also spawned a new research field **-Reconfigurable Computing-** based on the integration of variable hardware in general purpose computing.

1.1 Early Work

The first work on reconfigurable computing (RC) was done by Gerald Estrin in the 60's [51, 50, 49]. Estrin designed a system, the *fix-plus* machine, consisting of 3 elements: a GPP, a variable hardware and a supervisory unit. The fix-plus machine was intended to be used for accelerating Eigenvalues computation of matrices [51]. The available technology at that time made the use of the fix-plus machine difficult. Reconfiguration had to be done by hand, and substantial software efforts were required to implement applications. In the year 1977, Rammig [114] proposed a concept for editing hardware. Similar to the today's Field Programmable Gate Array architecture, the editor was based upon a set of modules, a set of pins and a one to one mapping function on the set of pins. A circuitry was then defined as a "string" on an alphabet of two letter (w

= “wired” and u = “unwired”). In order to build the hardware editor, selectors were provided with the modules output connected to the input of the selectors and the output of the selectors connected to the input of the modules. This structure was similar to that of switch boxes which are used inside Complex Programmable Logic Devices (CPLDS) and FPGAs. The introduction of commercial Field Programmable Gate Arrays by Xilinx in the mid-1980s [28] increased interest in reconfigurable computing. Since then, many experiments have been done and many reconfigurable systems have been built. The most popular experiments include the work on the programmable active memories (PAM) done at the DEC research center in Paris [126, 107], the SPLASH system of the Supercomputer Research Center, the work of Thomas Kean at the university of Edingburgh [92] and the XPuter of the university of Kaiserslauten [77, 76].
Many authors use different terminologies to categorize the various types of reconfigurable systems. A functional classification of reconfigurable systems can proceed as follows:

- **Application Specific Processor (ASP):** An ASP is a device tailored to compute a type of application at a given time. ASPs were the first machines to exploit the advantages of reconfiguration. They were built to perform special purpose computation and not intended to be very flexible. This programmable device is used for prototyping of ASIC function. The Ganglion and the RRANN [45] machines are examples of ASPs built for neural network computations. Other applications of ASPs include, but are not limited to statistical physics, embedded control and rapid prototyping.

- **Custom Instruction Set Processors (CISP):** For each application which has to be implemented on this system, an optimal set of instructions for the reconfigurable logic is synthesized. Most of them are used tightly with a GPP. The functions configured into the RPU are viewed by the host as other instructions available to the processor. It can be replaced indefinitely to accommodate new sets of instructions required by an application. Instructions can be dynamically replaced by new ones when there is no more need. Among those systems are, the Processor Reconfiguration trough Instruction Set Metamorphosis (PRISM) [10], the Xputer [76, 77], the Dynamic Instruction Set Computer (DISC) [128]

- **Reconfigurable Supercomputer (RS):** These are machines with a large amount of RPUs connected together in a stand alone system. They are normally connected to a host processor by a high bandwidth bus. A large amount of memory is available on those systems. Reconfigurable Supercomputers differ from ASPs only in their scale and the size of applications they implement. Among those systems we can cite the programmable active memory (PAM) developed by DEC [126, 107]. The PAM has been successfully tested and provided the fasted Rivest-Shamir-Adleman (RSA) Cryptography implementation to that date [126]. We can also cite the SPLASH [24], a systolic array developed by the Supercomputing Research Center in 1988. SPLAH connects up to 16 boards with an array of up to 16 FPGAs on each board. The second version, the SPLASH II has been successfully used in many Applications including searching of a genetic database, fingerprint and text and image processing.

- **Reconfigurable System on a Chip (RSoC):** A RSoC incorporates a processor, a memory and a reconfigurable unit on the same die. The first concept for a RSoC was provided in the Garp [78] architecture and the NAPA 1000 [5]. The first RSoCs on the market were designed for embedded control processing. On the Atmel FPGA, a processor with a few megahertz was connected to a small size FPGA and a small size memory. Rapid advancements in technology have allowed for the immersion of hard core processors with up to a few hundred megahertz today inside a device with millions of gates. This provides more than 10 times the performance of a reconfigurable supercomputer on a chip. The most recent examples of

high performance RSoCs are the Excalibur system of Altera [86] and the Xilinx Virtex II Pro [4]. In the Xilinx Virtex II pro, up to four 32-bits IBM PowerPC hard-cores with 300 MHZ are immersed in the chip in such a way that the placement of other modules on the FPGA surface device can not be disturbed. The ARM-based Excalibur of Altera features one RM922T processor with 200 MIPS performance and integrates on board memory, external interfaces and standard peripherals.

1.2 Challenges

Most of the successful experiments done with FPGA were limited to the area of Rapid Prototyping, focusing on the porting of functions already implemented for ASIC on the FPGAs. The devices are configured once before the execution of an application. The configuration remains until the end of the execution. While FPGA performance and flexibility have been aptly demonstrated, [76, 77, 126, 107, 5], reconfigurable computing has not taken great advantage of these strengths. Reconfigurable computing is characterized by the integration of the general purpose computing paradigm and reconfigurability in a computing system to increase performance and the flexibility. Thus, a typical reconfigurable system will consist of a processor and a reconfigurable device to work in close cooperation. The processor will not only be used to reconfigure the reconfigurable device and to move data to and from the reconfigurable device, but it will compute in parallel on a different set of data. The role of a reconfigurable device should not be relegated to that of a static device which is configured once for each application. The functions implemented in the reconfigurable device should also change with time to accommodate new functions. The implementation of functions on such systems requires a viable design methodology and viable compilers. Further, classes of applications that could benefit from such technology should be identified and implemented. Although hundreds of reconfigurable systems exist around the world, it is not always easy to find examples in which the computation takes place on both the processor and the reconfigurable device, and where the reconfigurable device is partly or fully reconfigured during computation. This is due to many factors:

- **lack of an appropriate design methodology :**
  The difficulty of understanding both the design process and reconfigurable systems programming is a significant barrier. The capabilities offered by such systems remain unknown by people unfamiliar with hardware programming. The implementation of algorithms in FPGAs is usually done in an ASIC-like fashion. The programmer starts with a specification of the algorithm in a Hardware description language (VHDL, Verilog, etc). The design has to be compiled, synthesized, simulated, placed and routed and finally the produced configuration, or bitstream, is downloaded in the FPGAs. In order to implement efficient designs, issues like clocking, pipelining, reconfiguration overhead have to be considered. If the FPGAs are integrated in a reconfigurable computing environment in which they compute in parallel to the GPP, then a synchronization mechanism should be considered between the processor and the FPGA. While FPGA-design time remains drastically shorter then ASIC-design time, implementing a function in FPGA can still take days, weeks, or even months. This is not acceptable for a software programmer or a mechanical engineer, who is used to implementing applications on a general purpose computer in few minutes or hours with far less difficulty and knowledge than required by FPGA programming tools. Applications for which a run-time reconfiguration is required are those which are too big to fit in the FPGA device. They must be partitioned in segments which will be successively executed in the FPGA. This process is known as **temporal partitioning** for non partial reconfigurable FPGAs and **temporal**

placement for partial reconfigurable device. Since very little investigation has been done in temporal partitioning and temporal placement, they represent a great challenge for the future of reconfigurable computing.

- **lack of appropriate compiler :**
  Ideally, we wish to provide potential programmers of reconfigurable systems with a universal environment in which all the difficulty of hardware programming is hidden. In such an environment, when a program is written in a high level language like C and C++, an equivalent workable and easy to modify code for a given reconfigurable system should be automatically generated. The user will program for a reconfigurable architecture without having to deal with issues like hardware/software partitioning, task distribution, simulation, timing analysis and hardware reconfiguration. The system should do the job for the user. For a distributed reconfigurable system[1], the generation of a parallel program from a sequential program as well as the task distribution on the computing elements have to be considered. Systems which offer the capabilities listed here are almost non existent.

- **difficulty to modify existing configurations :**
  The current FPGA design methodologies have been developed for ASIC implementation. They have the drawback of generating only fixed designs for a variable hardware structure. The designer cannot modify his design without running the complete steps (HDL-Synthesis-Place and Route), a process which is too long and difficult to be used in run-time reconfigurable environments. We would like have the possibility to modify our configurations quickly and with less effort. In order to be able to partially reconfigure the device, programmers should be able to select the part of the device to be replaced at a particularly time. The design should be placed and routed in such a way that the replacement of a part of the running code should not affect the rest of the design. That means modules to be replaced should not share a surface with other modules, a condition which requires great efforts to be fulfilled when implementing FPGAs with common design tools.

- **high reconfiguration overhead :**
  Until now, very few systems have implemented run-time reconfiguration [82, 66]. Apart from the lack of compiler and design methodologies, the reconfiguration time of the device has been a great bottleneck. Due to the large amount of data to be downloaded into the FPGA, the time needed for full reconfiguration is too high compared to the computation time, thus making the use of FPGAs for reconfigurable computing difficult. No great investigation of the partial reconfigurable devices has yet been done in order to reduce the reconfiguration overhead and increase the overall performance of reconfigurable devices.

- **difficulty to identify application domains :**
  Although the range of applications that could benefit from the reconfiguration is large, few experiments have shown a workable system integrating computation and reconfiguration [82, 66]. The inability for researchers to provide applications which justify the use of FPGAs in reconfigurable computing has hindered investigations in compilers and slowed down the development of tools to ease the implementation on reconfigurable systems.

The points previously mentioned represent big challenges in reconfigurable computing for which we provide our contribution in this thesis.

[1] A distributed reconfigurable system is a one system in which many reconfigurable platforms are connected together in a network.

1.3 This Thesis

Our contribution in the field of reconfigurable computing is done in the development of various algorithms to solve the temporal partitioning and temporal placement problem and the development of a simple and useful design environment for reconfigurable computing.

- **Temporal partitioning and temporal placement :**
  Functions that are too large to fit in one FPGAs are partitioned in different sections which are successively downloaded into the FPGA in accordance with a predefine schedule. Temporal partitioning normally targets non partial reconfiguration devices. Temporal placement defines for each module the time at which it will be mapped into the FPGA for computation. Additionally the position of the module inside the FPGA is given. Using partial reconfigurable devices, parts of the device can be replaced while the remaining part is still active. This is useful for example in systems which have to implement many modules at different period of time on the same device. Modules should be exchanged without disturbing the rest of the design. For temporal partitioning and temporal placement, efficient algorithms are required. In this work, we provide different solutions for each of those two problems as well as criteria for choosing the appropriate one.

  – **Temporal partitioning :**
    We have developed two methods to solve the temporal partitioning problem and provide motivations and reasons for choosing one of the solutions. The first one is an enhancement of the well known *list scheduling* method. A DFG is first partitioned using a list scheduling approach in which we introduced an area optimization step. The enhancement selects the next modules to be placed on the FPGA on the basis of the modules already assigned. The goal is to maximize the coexistence of two consecutive partitions on the device (configuration switching). After the partitioning process, a two dimensional *spectral placement* method is used to assign modules at their definitive locations on the FPGA. The second method uses a three dimensional spectral placement to position the modules in a three dimensional vector space. The partitioning is done by picking component along the time-axis in increasing order of the time-coordinates. Since we use an incremental partitioning method which does not guaranty a cycle free configuration graph, the Kernighan-Lin (KL) [93, 53, 98] algorithm is used to move the nodes of the graph from one partition to the other to insure an unidirectional partition. We introduce a modified gain function adapted to our need. The goal is not a minimum cut size like it is the case in many KL-FM algorithms, but a partition with all the edges converging in the same direction.

  – **Temporal placement :**
    Since partial reconfiguration is device dependent, we first take a look on the practice of partial reconfiguration on the Virtex FPGA [4] that we target in this work. This helps us define an implementation scheme based on a *first-fit* placement of clusters on the FPGA on the time-axis. We then provide two methods to compute the clusters of components to be placed. The first one is a level assignment approach similar to the list scheduling partitioning method and the second is a spectral clustering method. With the spectral clustering method, the components of the given DFG are first mapped into a 2-D vector space. A *cluster growth* approach is then used to build the clusters which represent the partitions needed to reconfigure the FPGA. With this method, connected modules are placed in the same packets which are used to partly reconfigure the device. With our method, it is not only possible to compute a temporal placement at compile-time, but

also at run-time. Once a temporal placement is computed, a sequence of packets is generated for the sequence of configurations. The objective is to compute a sequence of packets with minimum weight for a given function.

  - **Evaluation :**
    We define a measure of quality and use it to evaluate our different methods on a benchmark of randomly generated graphs.

- **Simple and useful design environment :**
  We have developed the **CoreMap design environment** [20] for an easy and fast programming of FPGAs. We follow a coarse grain approach to specify algorithms to be implemented. Operators, data and control sequences build the kernel of our components. Intellectual property (IP) cores implement operators like adders, multiplier, etc.... Control sequences are implemented using state machines, comparators and multiplexers.
  At the moment, the CoreMap provides support for the Xilinx Virtex FPGAs but can easily be extended to target many other FPGA devices. A design can be captured or loaded in a graphical editor. Modules can be selected from a library, placed and connected together on a graphical representation of the FPGA surface. A fast synthesis of the DFG permits the generation of configurations to program the FPGAs in few seconds. With this, it is possible for the designer to modify the design and quickly generate a new configuration. CoreMap also integrates our temporal partitioning and placement methods. Therefore a large design can be partitioned and successively emulated on one FPGA.

1.4 Structure of the Work

The rest of the thesis is organized as follow: Chapter 2 presents the architecture targeted in this work and presents a computation flow in which the FPGA and the CPU compute in parallel to solve a given problem. In chapter 3 we provide the definition of terms which are used in the thesis. We also provide background and formally state the problems to be solved in this thesis. The related work is the subject of chapter 4. We present the work of various authors in the area of temporal partitioning and temporal placement and highlight the advantages of each method as well as the differences between the different methods. Chapter 5 represents the main part of the thesis. In this chapter, we present the methods which were developed to solve the temporal partitioning and temporal placement problem. The result of the implementation of the different method on a benchmark of randomly generated graphs are presented in chapter 6. The algorithms developed in chapter 5 are implemented in the CoreMap design environment that we describe in chapter 7. Chapter 8 concludes the work and gives some indications of future work.

Chapter 2

Target Architectures

This chapter provides a brief overview on the FPGA technologies as well as the implementation methodologies. We present the target architecture and propose a computation flow in which the processor is not only used to move data to and from the FPGA, but also to compute with the FPGA in parallel on a different set of data.

2.1 Field Programmable Gate Arrays

2.1.1 Introduction

Time to market as well as time on the market is crucial in the electronic design business. Electronic manufacturers should bring their products as fast as possible on the market in order to amortize the development costs. On the other hand, electronic devices should remain on the market as long as possible. This requires a fast elaboration of prototypes and the possibility to modify the behavior of existing systems. FPGAs have emerged as the ultimate solution to reduce the time to market of electronic products. The supply of skilled hardware programmers capable of dealing with the complexities of FPGA implementation, constrains the number of potential FPGA applications. Broad applications are possible, as FPGAs can be used to implement systems which change their behavior to adapt to new environments. FPGAs can be also be used as stand alone devices to control a given process, or they can be coupled with other computation modules like processors in a given system.
FPGAs are a hybrid solution between programmable logic arrays (PLA) and mask programmable gate arrays (MPGA). Like PLA they are fully electrically programmable by the end user. Like MPGAs they can implement very complex functions. Since their introduction in 1985 by the Xilinx company [28], their capacity has increased from a few hundred gates in the last decade, to a few million gates today. The best known varieties of FPGA technologies are the anti-fuse and the SRAM based FPGAs. In the first case, special anti-fuses are included at each customization point. The two-terminal elements are normally disconnected, but by applying a high voltage, the terminals permanently connected. Since ”blowing” an anti-fuse should be an infrequent operation, anti-fuse based FPGAs are not suitable for devices which must be frequently reprogrammed, as is the case in reconfigurable computing. The second and more widely used type of FPGA is SRAM based. SRAM bits are connected to configuration points in the FPGA. The FPGA is configured or programmed by setting the desired value of the SRAM to match a given boolean function. In this way, a SRAM based FPGA can be programmed indefinitely. It is the SRAM based FPGAs that is the focus of this work.

2.1.2 Architecture

An FPGA is an array of processing elements called a configurable logic block (**CLB**) which can be connected via an array of programmable interconnection elements (Fig 2.1).

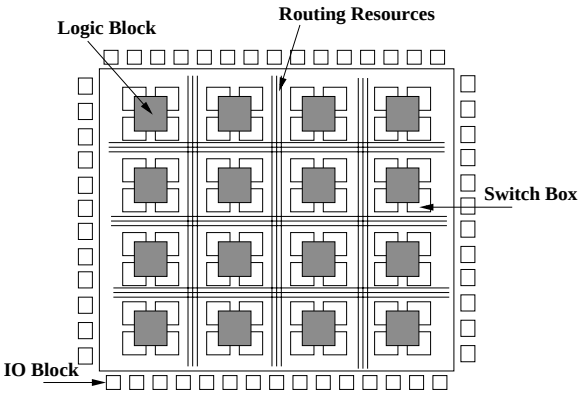


Figure 2.1: Structure of a FPGA [34]

Configurable Logic Block (CLB)

For the sake of simplicity, we've standardized on the Xilinx terminology (CLB) for the processing elements. This terminology can vary from vendor to vendor, but the meaning is almost the same. A CLB is constructed from the following components:

1. **Look up tables (LUT):** A CLB contains a certain number[1] of LUTs that are the basic computing elements inside FPGAs. An n-input LUT is an n-address memory used to store the 2^n possible values of an n-inputs boolean function. With a n-input LUT it is possible to implement any function with n variables. The values of the function for any combination of the n variables is computed and stored in the LUT. The actual variables are used to address the LUT at the location where the correct value is stored. The result appears at the LUT output.

2. **Flip Flops (FF):** Flip flop are used to temporally store values. The value to be store in the FF can be the LUT-output or a signal with an external source.

3. **Multiplexers (MUX) :** Multiplexers are used in CLBs to connect the LUT-output or another CLB-input signal to the FF-input or to a CLB-output

Fig 2.2 shows the Xilinx Virtex CLB which is used as reconfigurable processing unit (RPU) in this work.

Programmable Interconnections

The routing architecture of FPGAs consists of horizontal and vertical wires to connect the inputs and outputs of CLBs in different rows and columns. Additionally switch boxes exist to allow the programming of interconnections (Fig 2.1). The CLBs are embedded in the routing structure. The switch boxes provide programmable multiplexers, which are used to select the signals in the given routing channel that should be connected to the CLB terminals. The switch boxes can also connect vertical and horizontal lines, thus making a routing possible on the FPGA.

[1] Typically 2 or 4 inputs LUT

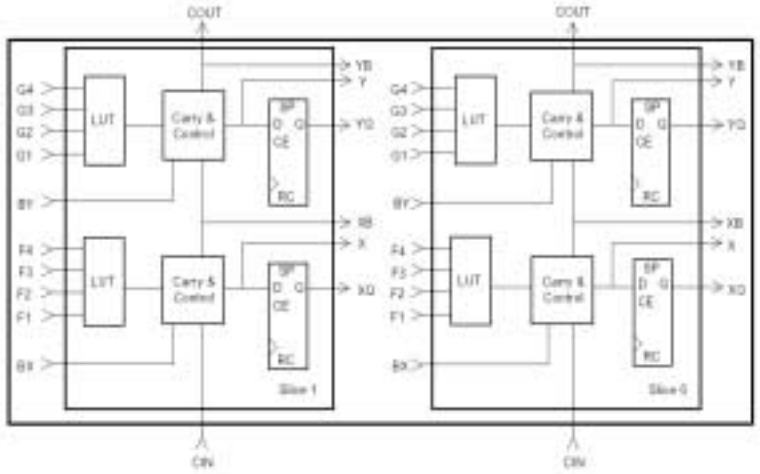


Figure 2.2: A Xilinx Virtex CLB [2]

2.2 Coupling

As explained in section 1.1, reconfigurable devices are often coupled with a host processor. The ”distance”[2] between the host processor and the reconfigurable hardware defines the frequency of the reconfiguration. On one end, we have systems which are designed to be used in stand alone mode. They are not frequently reconfigured. The host processor is used only to download the bitstream in to the reconfigurable device. Once the bitstream has been downloaded into the reconfigurable device, the system is decoupled from the host processor and the reconfigurable device operates as an ASIC in the new environment. Often the bitstream is kept in an EPROM from where it is downloaded in the FPGA once the system is switched on. An example of such system is the rapid prototyping environment for mechatronic systems RABBIT [106]. On the other end are systems in which the FPGA is frequently accessed by a host processor. The host processor is not only used to reconfigure the FPGA, but also to manage the data and functions to be computed in the FPGA. Since the FPGA is viewed as a kind of function accelerator, part of the FPGA can be reconfigured to accommodate new instructions and functions in the FPGA. Registers are used by the host processor to access the FPGA and temporally hold the results of computations, when the device is reconfigured. This is the case in systems like the RC-1000 of Celoxica [1], the RAPTOR [88] and the Spyder [47]. Between those two coupling strategies exist a large number of coupling strategies in which the “distance” between the GPP and the FPGA depend on the functionality of the system.

2.3 Hardware Software Partitioning

For systems that integrate both, a GPP and a reconfigurable hardware, the task of the system must first be partitioned into two parts. This process is called hardware-software partitioning or hardware-software co design. One part will be executed on the GPP and the other on the reconfigurable hardware. In general, complex control sequences are executed on the GPP while data path operations are more optimally executed in hardware. A considerable amount of work has been done in the past in the area of hardware-software co design [71, 72, 74, 73, 89, 23]. The hardware-software partitioning can be done manually or automatically by appropriate compilers [113]. Hardware-software partitioning is not within the scope of this work. We assume that this

[2]The term distance is used to express how tightly a processor can cooperate with the FPGA during a computation

step has been done and that sections of the program to be implemented in software are available, and the task to be executed on the reconfigurable device is available as a DFG. For the latter, the following design flow is used in general to program the FPGA.

2.4 Design Flow

The most used methodology to implement FPGA designs is borrowed from the ASIC design flow. The ususal steps are presented below:

2.4.1 Design Entry

The description of the function is made using a schematic editor, a hardware description language (HDL), or a finite state machine (FSM) editor. A schematic description is made by selecting components from a given library and connecting them together to build the function circuitry. When using a HDL like VHDL or Verilog, which are the most established HDLs, the behavioral description of the circuit can be done using the available constructs. Control dominated parts of functions can be entered using a FSM editor, in which states, inputs and outputs as well as the behavior of the FSM can be described.

2.4.2 Functional Simulation

After the design entry step, the designer can now simulate the design to check the correctness of the functionally. This is done by providing test patterns to the inputs of the design and observing the outputs. The simulation is done by tools which emulate the behavior of the components used in the design in software. During the simulation, the inputs and outputs of the design are shown on a graphic interface which describes the signal evolution in time.

2.4.3 Logic Synthesis

After the design description and the functional simulation, the design is compiled and optimized. It is first translated into a boolean equation. Technology mapping is then used to implement the function with the available modules in the target architecture. For FPGA, this step is called LUT-based technology mapping, because LUTs are the modules used in the FPGA to implement the boolean operators. The result of the logic synthesis is called a netlist description of the function. The net list of a function describes the modules used to implement the function as well as their interconnection. There exist different netlist formats to help exchange data between different CAD tools. The most known are the Electronic Design Interchange Format (EDIF) and the Xilinx Netlist Format (XNF) for the Xilinx FPGAs.

2.4.4 Place and Route

For the netlist generated in the logic synthesis process, operators (LUTs and FFs) should be placed on the FPGA surface and connected together via routing. Those two steps are normally achieved by CAD tools provided by the FPGA vendors. After the placement and routing of a netlist, the CAD tools generate a file called a bitstream. The bitstream provides the description of all the bits used to configure the CLBs and the interconnection switch boxes of the FPGA.

2.4.5 Configuration

The FPGA is configured by setting the corresponding bits in the LUTs, FFs, and by setting the right values for the multiplexers. This is done by downloading the bitstream produced in the preceding step into the FPGA. This can be done via an IEEE Standard 1149.1 JTAG (Joint Test Action Group) [4] boundary-scan interface or using another interface like the Peripheral Component Interconnect (PCI). The former is used for systems in which the reconfigurable device is decoupled from the host processor and the latter applied for systems in which the reconfigurable logic is tightly coupled to the host processor.

2.4.6 Design Tools

The design entry, the functional simulation and the logic synthesis are done using the CAD tools from Xilinx, Synopsys, Synplicity, Cadence, HSPICE, ALTERA MAX+II and Menthor Graphics. For the place and route, tools from Xilinx, Synplicity, Menthor Graphics can be used, but vendor tools are preferred. The format compatibility between the logic synthesis tools and the vendor's place and route tools makes the translation from one tool to the other sometimes difficult, thus increasing the learning time for FPGA implementation.

2.4.7 Processor/FPGA Computation Flow

In many experiments done in reconfigurable computing, the host processor was used only to configure the FPGA and control the dataflow between the FPGA and the processor [76, 77, 126, 107, 5]. This is not an efficient way to implement these systems, since the powerful processor is in an idle state is most of the time. We wish to have the host processor and the FPGA computing in parallel on different sets of data. For this purpose, we propose the computation flow illustrated in Fig 2.3.

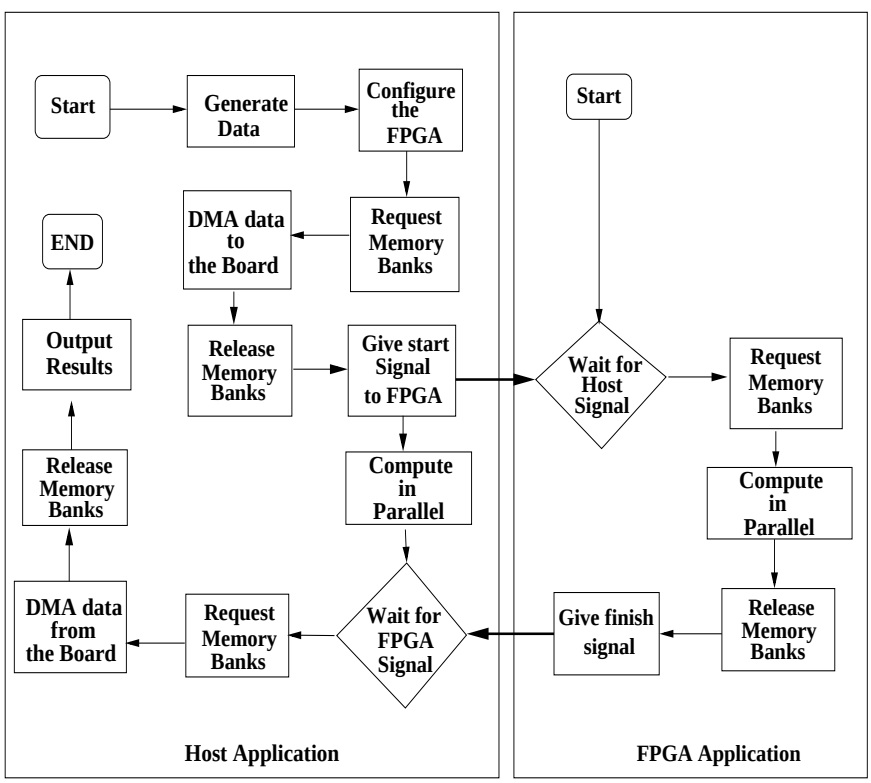


Figure 2.3: Host-FPGA computing paradigm

At the beginning of a computation, the host processor configures the FPGA. Then it requests on board memory and downloads the segment of the data to be processed by the FPGA. The memory banks are free to allow the FPGA to access data for processing. Via the control port of the FPGA, the host sends a signal to the FPGA to start processing. From then on, the host and the FPGA can process their segments of data in parallel. At the end of its computation the host reads the finish signal of the FPGA from its status port. The data can now be collected from the on board memory by the processor. A barrier synchronization mechanism should be implemented on the control and status port. By writing on the control port or reading from the status port of the FPGA, the processor waits until the FPGA has read the signals from the control port or written the signals to the status port. Data transfer from the processor memory to the on board memory and vice versa is done by DMA transfers. In the computation flow presented here, the FPGA is configured only once. The main steps for the processor are:

1. Start
2. Configure the FPGA
3. Download Data for FPGA computation into on-board memory
4. Computes in parallel with the FPGA
5. Upload the data computed by the FPGA from the on-board memory
6. stop

If the FPGA has to be configured more than once, then steps 2 to 5 should be repeated according to the number of reconfigurations to be done before step 6.

2.5 Target Architecture

2.5.1 Initial System

When we started working with FPGAs, we had a global view of what level of performance could be achieved with these devices. Meanwhile, a decision had to be made as to which type of device to work with. The idea was to connect FPGA boards on each node of a cluster of workstations. Because the communication overhead between nodes in such a cluster can affect performance if the number of processors increases too much, our aim was to reduce the communication level between processors, and to increase computing power at the node level. FPGA reconfigurability enables us to balance the computational load of time consuming parts of a program across a number of FPGAs. To test this, we attached sixteen FPGA boards on eight nodes of a cluster. After the implementation of two applications, we quickly realized the limits of this approach. We discovered that that our high performance computation goals were not be easily reached because of the following reasons.

- **Memory :** The amount of on board memory was too small to hold the results of some computation. This is very important in vector or matrix computation. Much of the time, a stream of data has to be modified by the same function implemented in the FPGA. This requires sufficient memory to hold the computation results. The data can then be collected by the FPGA from host memory, computed in the FPGA and stored in on-board memory. Finally, results are written back to the host memory.

- **Reconfiguration overhead :** As partial reconfiguration was not supported, the reconfiguration overhead of our FPGAs was too high. Configuration can only be done by downloading a complete bitstream to replace the old one inside the FPGA. This process is time consuming, because the FPGAs has to be reset before the download and a large amount of configuration data had to be downloaded trough a serial port.

Because of these two limitations we decided to change our platform by purchasing FPGA boards with a reasonable amount of memory and partial reconfiguration support. We purchased the Celoxica RC1000-PP board [1] to build a one node computing system. The decision to choose a board containing a Xilinx Virtex FPGA was done because of the granularity of those devices. Using fine grained CLBs like the Atmel FPGA would have increased the complexity of the functions to be implemented and slowed down the synthesis and mapping processes. Because we follow a core based design approach, a coarse grained FPGA best fits our needs. Superior architecture, and support for partial reconfiguration drove our board selection decision. In fact, the RC1000-PP is one of the few boards on the market that supports partial reconfiguration. The result of the experiment on our one node system can be applied to a cluster by replication of the actual system.

2.5.2 Current Experimental Platform

Our current computing platform is based on a Celoxica RC1000-PP board embedded in a personal computer. The board is connected to the processor via a PCI bus. The RC1000-PP features one Xilinx 1000 FPGA and 4 memory Banks of 2 Mbytes each. This is not too much, but sufficient to hold for example a 1000 by 500 dense matrix or a million by million sparse matrix with 24% non zeros. Furthermore, the four memory banks around the RC1000-PP can be used to increase the computation on four sets of data in parallel by four different modules inside the FPGA. With the computational paradigm presented here, the host application is implemented in C/C++. For the FPGA implementation we use the **DK1** [84] design environment of Celoxica, a Handel C design environment. Functions are implemented in Handel-C [85], a C-like design language which provides some features to handle parallelism, channels and interfaces. From a Handel-C description of a FPGA application, the DK1 synthesizes the design and generates a netlist in an EDIF format. The netlist is then passed to a vendor place and route tool which generates a placement and routing, and finally a bitstream to configure the device. Because of its high abstraction level, implementing an application in Handel-C is easier and faster than doing it in VHDL. As is the case with VHDL compilers, the DK1 generates a netlist in which LUTs and the flip-flops represent operators. The position of the different LUTs and flip flops are defined by the place and route tool. This can lead to the loss of regularity structure of the FPGA during the implementation of coarse grained functions like adders and multipliers.

Chapter 3

Background and Definitions

This chapter provides some backgrounds and general definitions of terms which will be used in the rest of the thesis. From the architecture of our target FPGAs, we derive definitions and state the problems to be solved in the remainder of the thesis.

3.1 General Definitions

Definition 1 (Dataflow Graph) *Given a set of tasks $T = \{T_1, ..., T_k\}$,*

- *a dataflow graph is a directed acyclic graph $G = (V, E)$, where $V = T$ is the set of nodes and E is the set of edges.*
- *An edge $e = (v_i, v_j) \in E$ is defined through the (data)dependence between task T_i and task T_j.*

A DFG represents a function to be implemented inside an FPGA. A task T_i in the DFG is implemented as a *core*[1] C_i.

Definition 2 (Latency, length, height, area, weight of nodes and edges) *For a node $v_i \in V$ with length x_i and height y_i and an edge $e_{ij} = (v_i, v_j)$,*

- *x_i denotes the length and y_i the height of v_i.*
- *a_i denotes the area of v_i.*
- *The latency t_i of v_i is the execution time of v_i in the FPGA.*
- *w_{ij} defines the weight of e_{ij}. It defines the width of BUS connecting two components v_i and v_j.*
- *The latency t_{ij} of e_{ij} is the time needed to transmit data from v_i to v_j.*

Definition 3 (Graph Connectivity) *Given a DFG $G = (V, E)$, we define the* **connectivity**, $con(G) = \frac{2 \times |E|}{|V|^2 - |V|}$, *of G as the relation of the number of edges in E over the number of all edges which can be built with the nodes of G.*
For a given subset $V\prime$ of V, the connectivity of $V\prime$ is defined as the relation of the number of edges connecting the nodes of $V\prime$ over the set of all edges which can be built with the nodes of $V\prime$.

[1] The core we considered are those available in the JBits environment or in the Core generator of Xilinx.

The connectivity of a set provides is a mean to measure how strongly the components of a set are connected. High connectivity means a strongly connected set, while low connectivity reflects a graph in which many modules are not connected together. This will be used later to decide which algorithm to run, given an instance of a problem.

Definition 4 (Configuration) *Given a reconfigurable processing unit H and a set of tasks $T = \{T_1, ..., T_n\}$ available as cores $C = \{C_1, ..., C_n\}$,*

- *we define the configuration ζ_i of the RPU at time t_i to be the set cores $\{C_{i1}, ..., C_{ik}\} \subseteq C)$ running in H at time t_i. We set $\zeta_i = \{C_{i1}, ..., C_{ik}\}$*
- *The configuration is geographically characterized by the set of positions P_i $(= \{p_{i1}, ..., p_{ik}\})$ where p_{ij} defines the position[2] of the core Cij on H for $1 \leq j \leq k$.*

Definition 5 (Schedule and Ordering Relation) *For a given DFG $G = (V, E)$, a* **schedule** *is a function $\varsigma : V \rightarrow N$. A schedule ς is feasible if: $\forall e_{ij} = (v_i, v_j) \in E : \varsigma(j) \geq \varsigma(i) + t_{ij}$.*
We define an **ordering relation** *$\leq$ among the nodes of G. $v_i \leq v_j \iff \forall$ schedule ς, $\varsigma(v_i) \leq \varsigma(v_j)$. It is obvious that $\leq$ is a partial ordering, since it is not defined for all pairs of nodes in G.*

Definition 6 (Partition) *A partition P of the graph $G = (V, E)$ is its division into some disjoint subsets $P_1, ..., P_m$ such that $\forall P_k \subseteq P$:*

- $\bigcup_{k=1}^{m} P_k = V$
- $\sum_{v_i \in P_k} a_i \leq A_k$ *where A_k is a limit on the area of P_k.*
- $1/2(\sum_{\{e_{ij} \in E : (e_{ij} \cap P_k) \neq \emptyset \text{ and } (e_{ij} - P_k) \neq \emptyset\}} (w_{ij})) \leq T_k$ *where T_k is a limit on the terminals of P_k.*

We extend the ordering relation $\leq$ to P as follow: $Pi \leq P_j \iff \forall e = (v_i, v_j) \in E$ with $v_i \in Pi$ and $v_j \in Pj$, either $v_i \leq v_j$ or $\leq$ is not defined for v_i and v_j. The **partition P is ordered** $\iff$ *an ordering relation $\leq$ exists for P.*

The partition is subject to the constraint that the sum of the area of all elements in a partition as well as the sum of external edges (edge connecting modules in two different partitions) should not exceed a given limit. The area constraint is the size of the FPGA while the terminal constraint is the number of FPGA pins. An ordered partition is characterized by the fact that for a pair of partitions, one can always be implemented after the other with respect to any scheduling relation.

Definition 7 (Temporal Partitioning) *Given a DFG $G = (V, E)$ and a reconfigurable processing unit H, a temporal partition of G on H is an ordered partition P of G for the RPU H.*

If a set S $(= \{H_1, ..., H_d\})$ of identical devices H is available on which the partitions can be mapped and $d > 1$, then we have a **multi-RPU or multi-FPGA temporal partition**, otherwise it is a **single-RPU or single-FPGA temporal partition**. In a single-FPGA temporal partitioning, only one partition can be downloaded into the FPGA at a time, while in a multi-FPGA temporal partitioning many partitions can be downloaded at same time into the available FPGAs.

Definition 8 (Quality) *Given a DFG $G = (V, E)$ and a partitioning $P = \{P_1, ..., P_n\}$ of G, we define the quality $Q(P) = \frac{1}{n} \times \sum_{i=1}^{n} (con(P_i))$ of P as the average connectivity over all the partitions P_i $(1 \leq i \leq n)$.*

[2]The position of a core on a device is given by the coordinates of the lower left corner of it's bounding-box

The quality of a partition is a mean to find out how good a partition is for our purpose. If an algorithm assigns connected modules to the same partition, then the quality of the resulting partition will be high. Otherwise, the quality will be low.
We target architectures in which the FPGA is connected to a host processor by a bus like the PCI-Bus. The number of BUS lines is limited. Therefore, the communication has to be time-multiplexed on the bus if many data have to be transported on the processor-FPGA bus. This happens for example when a partition has to be replaced in the FPGA. Because the communication between the partitions is done by a set of registers inside the FPGA, all the temporary data in those registers have to be saved in the processor's address space before reconfiguration. The device is then reconfigured and the data are copied back into the FPGA registers. Our goal during the temporal partition will be to minimize the set of registers needed to communicate between the generated partitions. This goal is likely to be reached if highly connected components are placed in the same partition. After a partitioning by a given algorithm, the quality of the partition will determine if the algorithm performed well or not. If a graph is highly connected and the partitioning algorithm performs with low quality, then there will be more edges connecting different partitions and therefore more data exchange among the partitions. But if the graph is highly connected and the partitioning algorithm performs with high quality, then the components in the partitions are highly connected and therefore there will be fewer edges connecting the partitions. Figures 3.1 and 3.2 illustrate the connectivity of a graph and the quality of an algorithm. In figure 3.1 a graph with a connectivity of 0.24 is partitioned by an algorithm which produces a quality of 0.25. The same graph is partitioned by another algorithm in figure 3.2 with a quality of 0.45. In the first case, we have 6 edges connecting the two partitions, while there are only two edges connecting the partitions in the second case. The second case is, therefore, better than the first case for data communication between the partitions.

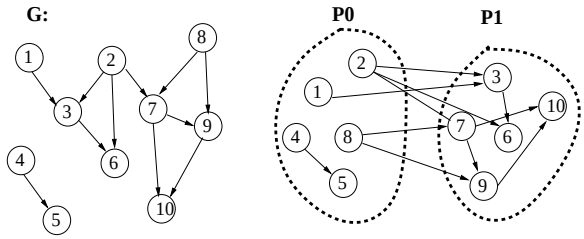


Figure 3.1: Partitioning of a graph with connectivity 0.24 and quality 0.25

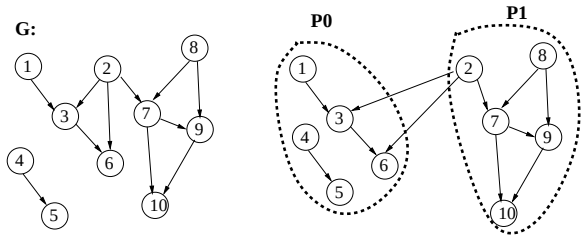


Figure 3.2: Partitioning of a graph with connectivity 0.24 and quality 0.45

Definition 9 (Configuration Graph) *Given a DFG $G = (V, E)$ and a temporal partition $P = \{P_1, ..., P_n\}$ of G, we define*

- *a Configuration graph of G relative to the partition P with notation $\Gamma(G/P)$ to be the graph $\Gamma(G/P) = (P, E^P)$ in which the nodes are partitions in P. An edge $e = (P_i, P_j) \in E^P \iff \exists e = (v_i, v_j) \in E$ with $v_i \in P_i$ and $v_j \in P_j$.*

- *Each node $P_i \in P$ has an associated configuration ζ_i that is the implementation of P_i for the given FPGA. The communication between the partitions is done via the interconfiguration registers (figure 3.3).*

The interconfiguration registers are registers inside the FPGA which are mapped in the processor address space. On configuration, the contents are saved on the processor address space. After reconfiguration, their contents are copied back inside the FPGA for further computation. The goal

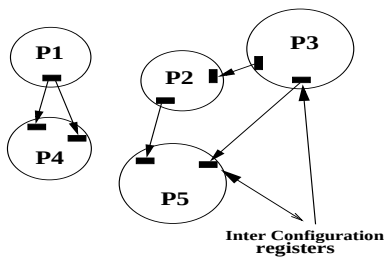


Figure 3.3: A Configuration Graph

of the temporal partitioning is the computation and scheduling of a configuration graph on one or many FPGAs. In this thesis we consider single-FPGA temporal partitioning. Graphs which contain bidirectional edges or cycles are not within the scope of this work. In the configuration graph, nodes represent the partitions and edges the external edges to the partitions. In the case of a multi-FPGA temporal partitioning, bidirectional edges and cycles can be allowed in the partition graph. In this case, the partitioning should be done with the additional constraints that nodes connected by bidirectional edges or nodes on the path of a cycle should remain in a group of partitions which can be downloaded at the same time into the available FPGAs.
In applications which require modules to be exchanged while the rest of the modules are still running, temporal partitioning is not adequate. Temporal placement that will be defined next should be applied in this case.

Definition 10 (Temporal placement) *Given a DFG $G = (V, E)$ and a device H with the size (h_x, h_y),*

- *a temporal placement is a three dimensional vector function $p = (p_x, p_y, p_t) : V \rightarrow N^3$ such that p_t defines a feasible schedule.*

- *The values $p_x(v_i)$, $p_y(v_i)$ and $p_t(v_i)$ denote the coordinates of the node v_i in the a 3 dimensional vector space. $p_x(v_i)$ and $p_y(v_i)$ define the coordinate of v_i inside the device H, while $p_t(v_i)$ defines the time at which v_i will be mapped in H.*

3.1.1 Application Domains

ASIC fabrication requires a lot of testing to ensure high quality and to guard against defects. Testing can be done by intensive simulation of the design. Simulation is usually accomplished by software tools that emulate the design. Large design simulation is normally too slow and insufficient to ensure final product quality. For this reason, simulation must be supported by emulation. Emulation speeds design testing because it uses real hardware. FPGAs offer the possibility of implementing a design and testing the hardware before production. The major problem with FPGAs emulation lies in size. Because ASICs are normally larger than than FPGAs, ASIC designs have to be partitioned across many FPGAs for emulation. Expensive equipment is required to realize emulation of large circuits, resulting in higher development costs for the final product. Using temporal

partitioning, the circuit can be "temporally" partitioned and the partitions can be used to program the FPGAs. In this case, the partitions are successively downloaded inside the FPGAs, and the complete circuit is computed in sequence. Temporal partitioning can therefore be used as a cheap solution in circuit emulation. In our environment in which the FPGA is embedded in a computer system, the circuit under test in the FPGA is accessed via input/output registers that are mapped in the processor address space. During the reconfiguration, the values of the input/output registers are temporally stored in the processor registers. Therefore, the need to minimize the amount of data exchange is high.
Temporal placement can play a great role in systems which require a great amount of flexibility. In mobile communication for example, the growing number of protocols and multimedia applications requires flexible, efficient and low cost devices. In order to integrate all the modules required in a system on chip, a large area is required for the devices. If an application allows only a small amount of modules to be executed at the same time, then chip area and power consumption can be saved. Using a reconfigurable device and a temporal placement algorithm will allow a large amount of modules to be executed on the same device. Temporal placement can also be very useful for systems for which a physical upgrade is impossible or too expensive. If we consider a system built on earth and stationed on another planet, the system offers no possibility for a physical upgrade. A reconfigurable device can play a great role here. The functionality of the system can be modified directly from the earth by sending a corresponding sequence of packets to the device. The device will partially be reconfigured without interruption of operation.

To illustrate the concept of temporal placement, let us consider an application taken from [25] and illustrated in figure 3.4. A number of objects moving on a conveyor belt must be recognized using a stereo vision system, consisting of two cameras. Suppose that the recognition process

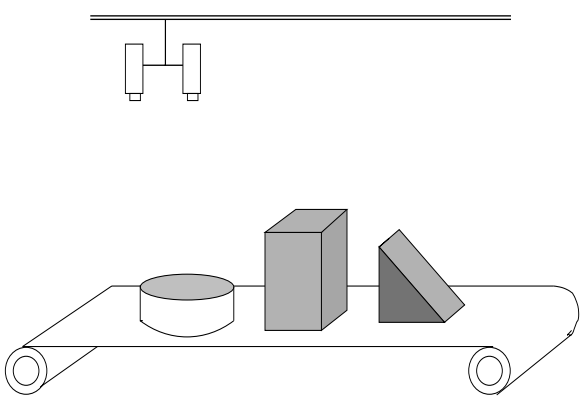


Figure 3.4: Industrial application which requires the visual recognition of object on a conveyor belt.

is carried out by integrating the two-dimensional features of the top view of the objects with the height information extracted by the pixel disparity on two images. As a consequence, the computational activities of the application can be organized by defining the following tasks:

- Two tasks *acq1, acq2* (each for one camera) dedicated to image acquisition, transfer images from the cameras to the memory.
- Two tasks *edge1, edge2* (each for one camera) dedicated to low level image processing. The operations performed here are digital filtering for noise reduction and edge detection.
- A task *shape* for extracting two-dimensional features from the object contours

- A task *disp* for computing the pixel disparities from the two images.
- A task *H* for determining the object height from the results achieved by the task *disp*.
- A task *rec* performing the final recognition.

¿From the logic relation existing among the computations, the precedence graph of fig 3.4 can be derived.
For this application, one partial reconfigurable device can be used to save space and power. Since the objects arrive on the conveyor over a given period, the device can be periodically and partially reconfigured to execute the different tasks. A possible temporal placement can then be computed as shown in figure 3.6 with the cores *acq*, *edge*, *shape*, *H* and *rec* on a partially reconfigurable FPGA. The FPGA is successively and partially reconfigured to execute two instances *acq1* and *acq2* of the core *acq* for the two cameras, two instances *edge1* and *edge2* of the core *edge*, one instance of the cores *disp*, *shape*, *H* and *rec*. With the computed temporal placement, the complete visual recognition application can be implemented in only one device. The modules will successively be downloaded onto the device according to the computed temporal placement.

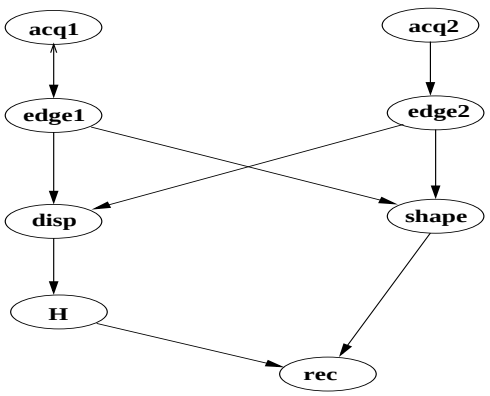


Figure 3.5: Precedence Graph of the corresponding industrial application.

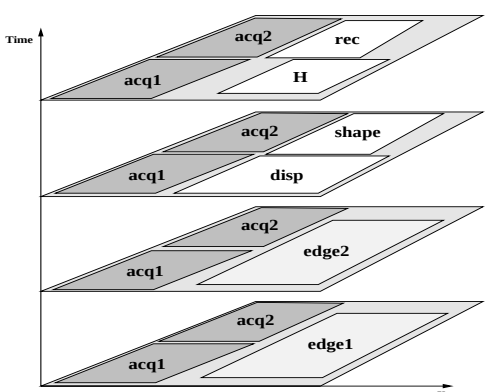


Figure 3.6: Temporal placement of the precedence graph of figure 3.5

Definition 11 (Some Operation on Cores) *Given a set of cores Ω, we define the following operations:*

- **Membership:** *A core C_1 is a member of a core C_2 ($C_1 \subseteq C_2$) iff the functionality of C_1 is implemented in C_2.*

- **Union:** *Union on cores is defined as follow:*
  $C_1 \cup C_2 = C_3$ *iff the functionality of* C_3 *can be obtained by merging the functionality of* C_1 *with that of* C_2.

- **Subtraction:** *Subtraction is the inverse of the union.* $C_1 - C_2 = C_3$ *iff the functionality of* C_3 *can be obtained by removing the functionality of* C_2 *from that of* C_1.

- **Intersection :** *Two cores* C_1 *and* C_2 *intersect if a core* C *exists such that* $C \in C_1$ *and* $C \in C_2$.

- **Size :** *The value* $A_C = C_x \times C_y$ *(with* C_x *being the length and* C_y *the height of the core* C*) is called the size of the core* C*. It is the area of the bounding box defined by* C.

3.2 Practical Considerations

In this section we will investigate the practice of reconfiguration. This will help us define the basis for our theoretical formulation of the problem to be solved in partial reconfiguration as well as the optimization to do on possible solutions.
The practice of reconfiguration is device dependant. It is therefore difficult to develop a general method for partial reconfiguration to be applied on each device type. We decided to consider the architecture of the Virtex FPGA in the development of our method, because it is one of the few coarse grained FPGA with large capacity to provide partial reconfiguration capabilities.
The partial reconfiguration support for the Virtex is provided by the tool JBits [63]. JBits permits arbitrarily small changes to be made directly to the Virtex device configuration data quickly and without interruption of operation. The Virtex FPGAs like other FPGAs are organized as a two dimensional array of CLBs containing a certain amount of logic. They are configured with configuration data called a **bitstream** which can be downloaded in the device. While many FPGAs do not allow partial configuration, the JBits-Virtex adopts a different approach [104, 3]. The idea behind partial reconfiguration is to realize reconfiguration by only making the changes needed to bring the device in the desired configuration. Fragments of the complete bitstream (**packets**) are sent to the device in order to reconfigure the needed part of the device. A copy of the last configuration is maintained in a dedicated part of the processor memory. This part of the memory is called the **configuration memory**. Partial reconfiguration is done by synchronization between the configuration memory and the device. Changes made between the last configuration and the present one is marked as packets which are then sent to the device for partial reconfiguration. A packet is a sequence of (command, data) pairs of varying length which are used to read or write internal registers and configuration state data. The Virtex is organized in **frames** or column-slices (figure 7.3). A frame is the smallest addressable unit. Because a frame occupies a column and the minimum width of the operators we consider is one CLB, we assume that the size of a frame is a column of FPGA's CLBs. In the following we consider a column and a frame as equivalent. In order to implement a reconfigurable application, packets needed during the computation will be recomputed by masking the differences between consecutive bitstreams. Sending a packet to the device for partial reconfiguration can be done in two ways:

- **Offline Schedule :** The sequence for sending the packets to the device is recomputed by a temporal placement algorithm.

- **Online Schedule :** Packets are requested and dynamically scheduled on the device for execution. The schedule is determined at run-time. In this case, many mechanisms like real-time

reconfiguration defragmentation and relocation of modules on the device can also be considered.

We propose the organization of a partial reconfigurable system illustrated in figure 3.7. A **configuration manager** is responsible for sending packets which are stored in memory to configure the device. On request, the configuration manager loads the packets needed to partially reconfigure the device and carry the reconfiguration by sending those packets to the device. In this work we are

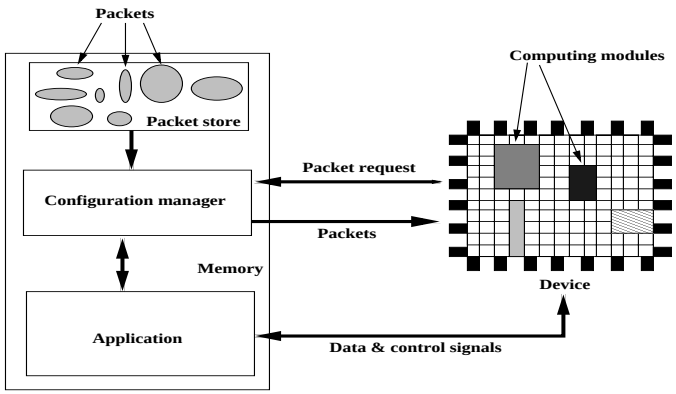


Figure 3.7: Model of a Partial Reconfigurable System

seeking the computation of the optimal sequence of configurations which is needed to implement a given set of tasks as defined in a DFG. We first provide a formulation of this optimization problem. Our approach for computing a solution will be provided in section 5.2. The definitions provided here rely on the Virtex architecture which was presented before.

Definition 12 (Packet) *Given a reconfigurable device H in which the smallest exchangeable part is one column or frame, we define a packet P_i to be the sum of all frames f_j or columns contained in the packet P_i.*

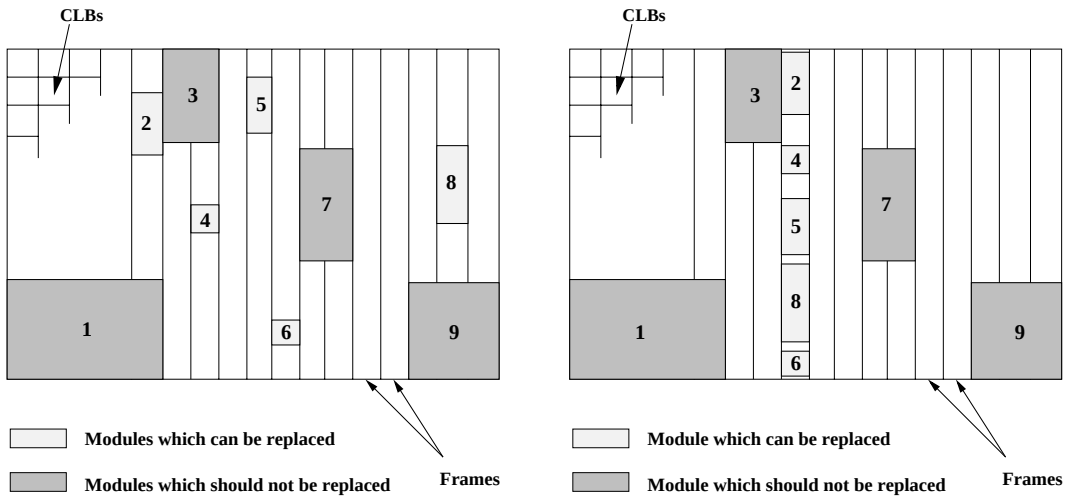
$$P_i = \sum_{j=0}^{k} f_{ij} \text{ with } f_{ij} \begin{cases} 1 \text{ if } f_j \in P_i, \\ 0 \text{ otherwise} \end{cases} \tag{3.1}$$

We also define the weight $w(f_j)$ of a frame f_j to be the amount of data in the frame f_j. The weigth $w(P_i)$ of a packet P_i is the sum of the weight of all frames in P_i. Because the amount of data in a frame is the same for all frames in a device, the weigth of a Packet can be seen as the number of frames in that packet.

$$w(P_i) = \sum_{j=0}^{k} w(f_{ij}) = K * n_i \tag{3.2}$$

Where K is a constant value for the weight of a frame and n_i is the number of frames in the packet P_i. A packet P_i as defined in this work is the amount of data necessary to move the device from the configuration ζ_i to configuration ζ_{i+1}. Therefore we have: $P_i = \zeta_{i+1} - \zeta_i$. With this, we can define the reconfiguration problem.

Definition 13 (Partial Reconfiguration Problem) *Given a reconfigurable processing unit H and a DFG $G = (V, E)$ where the set of tasks are available as cores $C = \{C_1, ..., C_n\}$, compute the sequence of configurations $\zeta_0, ..., \zeta_k$ which minimize:*
$\sum_{i=0}^{k-1} w(P_i) = \sum_{i=0}^{k-1} w(\zeta_{i+1} - \zeta_i)$.



(a) A non optimal placement of modules for reconfiguration

(b) Optimal mapping of modules to frames for reconfiguration

Figure 3.8: Clustering of modules on a partial reconfigurable device

Given a set of tasks to be implemented on a reconfigurable device, we seek a sequence of configurations which places the components to be reconfigured on the best locations. That means:

1. The placement of a component to be replaced produces the minimum amount of packets.
2. The replacement of a set of modules does not alter the work of the rest of components during reconfiguration.

To illustrate this, consider the two configurations of figure 7.3, in which the components 2, 4, 5, 6, 8 have to be changed. Replacing the components in the first case will lead to the replacement of five frames. Moreover, the operation of components 1, 3 and 9 will be interrupted during reconfiguration. In the second case the components 2, 4, 5, 6, 8 are placed in such a way that only one frame has to be exchanged and the operations of all other components are not disturbed. Reconfiguration is done by replacing a certain number of frames. If different modules located in different frames have to be replaced, all the frames containing those modules will be sent to the device as packets. This increases the reconfiguration overhead. Instead of sending many frames with only small valid parts, we seek the grouping of modules to be replaced in a minimum number of frames (figure 7.3).

Definition 14 (Packet By Frames Matrix) *For a sequence of configurations $\zeta = \zeta_0, ..., \zeta_k$, we defined the Packet By Frames Matrix of ζ as the matrix M in which the entry m_{ij} is either 0 or 1 and $m_{ij} = 1 \iff f_j \in P_i$. M is a $k \times n$ matrix where n is the number of the frames or columns in the given reconfigurable device.*

Our objective in partial reconfiguration is reduced to the minimization of the sum of all entries of the Packets by Frames Matrix over all sequences of configuration. Since all the entries in M are positive numbers, the partial reconfiguration problem is then equivalent to the *minimization of the 1-norm of the packet by frames matrix M*.
So far we have provided most of the definitions we needed in the work and explain the goal of our work. Before providing our methods to solve the problems stated in this chapter, pause to consider the work done by others in the past.

Chapter 4

Related Work

This chapter provides an overview of the previous work done in hardware reconfiguration in general. It presents various techniques and methodologies for temporal partitioning and temporal placement on partial and non partial reconfigurable FPGAs. Although the terms "temporal partitioning" and "temporal placement" are not always used by the authors, their works can be classified in those two categories. Some of the authors have developed methodologies for efficient reconfiguration and online temporal placement and proposed new FPGA architectures adapted to their methodologies.

4.1 Temporal Partitioning

In section 3.1 we provided a formal definition of the temporal partitioning problem and defined our goal as well as the quality of a partition in term of graph connectivity. In this section, we present some methods developed in the past by different authors to solve the temporal partitioning problem.

4.1.1 ASAP/ALAP-List Scheduling

The most used and perhaps the simplest approach used to solve the temporal partitioning problem is the list scheduling (LS) method [110, 27, 108, 30, 123, 112]. The idea behind the LS approach is first to place all the nodes of a DFG representing the problem to be solved in a list. A new partition (also called configuration) is built stepwise by removing nodes from the list and allocating them to the partition until the size of the partition reached a given size limit (the size of the FPGA). A new partition is then created and the process is repeated until all the nodes from the list are placed in partitions. The list is ordered either by an ASAP (As Soon As Possible) mechanism [112, 123, 110, 108, 44, 91] which puts a node in the list as soon as all it's predecessors have been placed in the list, or an ALAP (As Late As Possible) [110, 123, 91] paradigm which places a node in the list as soon as all it's successors have been assigned. Some methods combine the ASAP and ALAP in a precomputation step to determine the mobility range of the nodes [123] before applying optimization methods to select the best position of nodes in their interval range. The optimization process is problem and architecture dependant. In [110] for example, the goal is the minimization of the overall computation time of the given function through the generation of a minimum number of segments. For this purpose, a trade-off is performed between the number of segments and the latency of each segment during the exploration of the design space. A sharing of functional units is used to reduce the number of partitions. In [123], the optimization goal is the reduction of the number of LUTs, the number of nets and pins. After the list scheduling partitioning, a pair-wise

interchange is done between adjacent segments in order to minimize a cost function defined as the sum of squares of the number of nets in each segments. Pandey et al [110] and Chang et al [30] use an enhanced version of the force directed list scheduling algorithm of Paulin and Knight [111] to optimize their list scheduling based partitioning algorithm. Based on the probability of each node being placed at a specific time step, they compute a distribution graph that indicates the concurrency of similar operations. The force is then computed in such a way, that the operation that produces the lowest global increase in concurrency is selected.
The main advantage of the list scheduling method is its linear run-time. Many authors use this method to compute the mobility range of the components before optimizing their solutions. Furthermore the LS method allows for local optimizations while selecting the nodes to be placed in partitions. The main disadvantage of the algorithm is the *levelization* [1] effect of the partitioning. The connectivity in the original graph is not preserved in the partitions. This is due to the fact that modules are assigned to partitions based more on their level number, rather than their interconnectivity. With this, the goal of minimizing the number of nets connecting two different partitions, i.e. the minimization of the data exchange among partitions becomes difficult to reach. In figure 4.1 a graph is partitioned using the list scheduling method while in figure 4.2 the same graph is partitioned with a spectral method which better preserves the graph connectivity. The partitioning in 4.2 results in less communication among the partitions than the partition in figure 4.1.

Because of its linear time, the list scheduling algorithm remains a good temporal partitioning

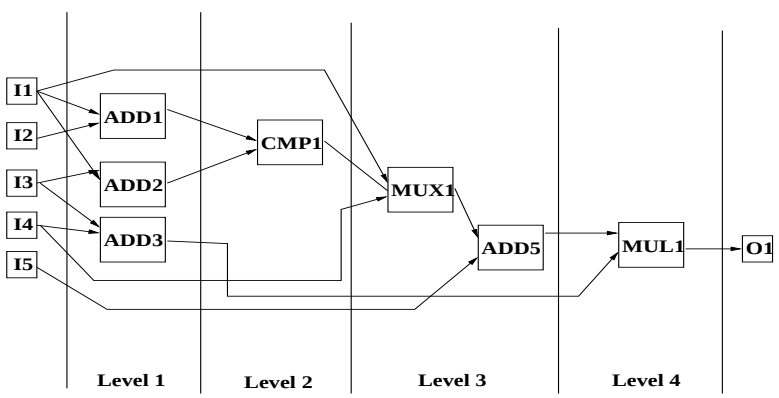


Figure 4.1: ASAP partitioning of a DFG

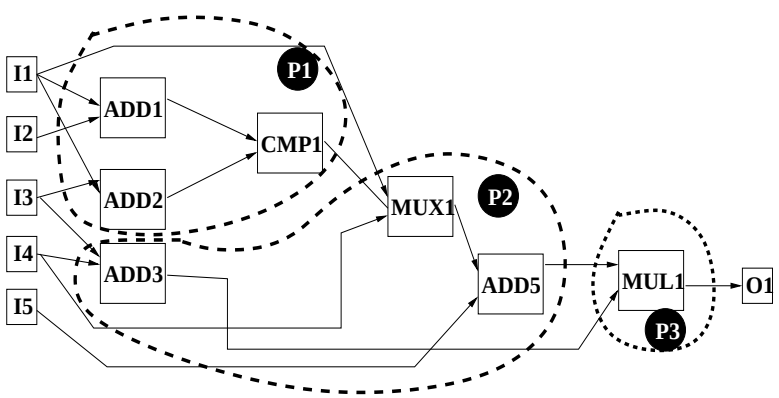


Figure 4.2: Spectral Partitioning of the DFG of figure 4.1

candidate, in particular for low-connected graphs. For this reason, we used it as a basis for our first

[1] Levelization means that the partitions are built on the basis of the level number of the DFG component.

temporal partitioning method and made some enhancements aimed at reducing the total number of generated configurations. While selecting the components to be placed in partitions, we do some local optimizations by allowing two configurations to reside on the device at the same time. This helps to reduce the number of configurations and therefore the overall computation time of the given function.

4.1.2 Integer Linear Programming

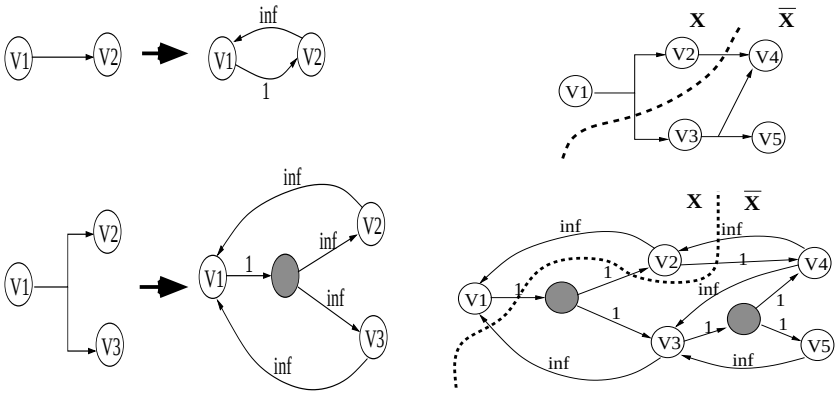
Integer Linear Programming (ILP) methods for temporal partitioning are used in [90, 91, 108, 44]. Kaul et al [90, 91] presented a combined ASAP/ALAP and ILP method. For a set of tasks to be partitioned for a given device, a fast list scheduling algorithm is used to estimate the upper bound of the latency and the size of a partition. Many 0-1 variables are then defined and several constraints are formulated as equations. The fundamental system modeling 0-1 variables are those which define the assignment of a task to a partition, the assignment of an operator to a functional unit, and the placement order of two variables in two different partitions. The constraints imposed are those aimed at defining a feasible schedule as well as the memory restrictions. The definition of the fundamental variables and constraints is derived from the fact that each task should be placed in a partition and that the placement should reflect the precedence order between two tasks as specified in the DFG. The objective is the minimization of the overall latency of the function to be implemented. The constraints and variables generate a 0-1 non-linear equation model which is solved using a linearization method. Similar to the former approach, the feasible constraints are defined by Ejioui et al [44] to determine the equations to be solved. The objective function is the minimization of the micro cycle [2] in a so called time multiplexed FPGA.
The general problem of the ILP approaches for partitioning a DFG is the size of the computation model which grows very fast and, therefore, the algorithm can only be applied to small examples [57]. To overcome this problem, some authors [95, 57] reduce the size of the model by reducing the set of constraints in the problem formulation, but the number of variables and precedence constraints to be considered still remain high. FPGAs are no longer small devices which could not hold more than four multipliers. Their sizes have increased very fast in the past and this will continue in the future. Temporal partitioning algorithms should therefore be able to partition very large graphs (graphs with thousands of nodes). Trying to formulate all the precedence constraints with the ILP approach will drastically increase the size of the model, thus making the algorithm intractable. On the other hand the authors requested that the amount of intermediate data stored between partitions should not exceed the available memory. This constraint is unlikely to help minimize the data exchange between partitions, since the processor will always have enough memory to store the data from the FPGA. Given the limitation that the goal of minimizing the data exchange between partitions cannot be satisfied by the ILP method, and that the algorithm becomes intractable for large graphs, ILP methods are not appropriate for solving our problem.

4.1.3 Network Flow

The network flow methodology is based on the Ford and Fulkerson min-cut max-flow theorem [54]. This method has been used in circuit partitioning by Yang [130] and Cong [83]. Liu et al [100, 99] applied the network flow method to find a solution of the temporal partitioning problem. The objective here is the minimization of the cut-size of the partitions as well as the minimization of the maximum partition size. As in the ILP methods, the precedence relation as well as the FPGA size defines the constraints for the algorithm. From a task graph G, a network graph

[2] In the so called time multiplexed FPGFA, the loading and execution time of a configuration is called a micro cycle

$G' = (V', E')$ is constructed as illustrated in figure 4.3 (a). The nodes of V are introduced in V'. For an edge $(v_1, v_2) \in E$ two edges $e_1 = (v_1', v_2')$ with capacity of $c_1 = 1$ and $e_2 = (v_2', v_1')$ with capacity $c_2 = \infty$, are added to E'. For a multi terminal edge in E, a bridging node is added



(a) Net modeling for a two terminal net and a multi-terminal net

(b) A cut in G' and the corresponding net-cut in G

Figure 4.3: The networkflow modell for temporal partitioning

to V'. An edge weighted with 1 connects the source node with the bridging node in E'. For each sink nodes in the multi terminal net, an edge weighted with ∞ is added between the bridging edge and the sink nodes and between the sink nodes and the source node. After the computation of a max-flow which produces a min-cut $(X', \overline{X'})$ of the network G', all the forward edges (from X' to $\overline{X'}$) must be saturated (flow equal to the capacity) and all backward edges (from $\overline{X'}$ to X') have zero amount of flow. If a net is cut, then only bridging edges can connect $\overline{X'}$ to X', thus preserving the precedence constraints. For a computed min-cut $(X', \overline{X'})$ in G', the corresponding cut $(X, \overline{X})$ is computed in G by inserting the equivalent nodes from $(X', \overline{X'})$ in $(X, \overline{X})$ (figure 4.3 (b)).
The min-cut max-flow theorem of Ford and Fulkerson is a powerful tool to minimize the communication in a cut. But the model is constructed by inserting a great amount of nodes and edges in the original graph. The resulting graph G' becomes too big[3] and difficult to handle, thus making the method not suitable to our purpose.
The three approaches (list scheduling, integer linear programming, and network flow)presented here all suffer from the limitation that their authors have primarily focused on the overall latency minimization of the design, while neglecting reconfiguration overhead and data exchange. In practice the reconfiguration overhead of FPGAs is very high. The reconfiguration overhead ranges from milliseconds to seconds depending on the type of port used. Neglecting such large values will lead to incorrect temporal partitions. Further, these approaches suffer from the drawback that they do not consider the geometrical properties of the modules to be mapped into the FPGA. In practice high level description languages and synthesis tools are used to specify and solve these problems. This leads to a non efficient use of the regularity structure of the FPGAs [26] and resources.

[3]In the worst case, the number of nodes in the new graph can be twice the number of the nodes in the original graph. The edges number of additional edges also grows dramatically.

4.2 Context Switching and Time Multiplexing

Reconfiguration time has always been a major problem in reconfigurable computing. Switching from one configuration to the next can take up to a few seconds, thus often making the reconfiguration time larger than the computation time in applications which reconfigure FPGAs [19, 110, 67]. In order to decrease reconfiguration overhead, the architectural concepts of time-multiplex and context switching FPGAs have been proposed[123, 117]. The general idea behind those two concepts is to store a given number of configurations directly on the chip. This avoids the downloading of a large amount of reconfiguration data when required. Time-multiplexed FPGA have been proposed by several authors. The best known one is the Trimberger's architecture [123]. Trimberger proposed an FPGA architecture in which designs are modeled as Mealy state machines. Micro registers are used to temporally hold computation data when switching from one configuration to the next. The combinational logic receives its inputs from the device inputs and from the flip-flop outputs, and the device outputs come from combinational logic and from flip-flops outputs. In *logic engine* mode, the device emulates a large design in many *microcycles*. In each micro cycle, resources are allocated to a new configuration. A similar architecture has been proposed by Scalera et al [117]. Data pipes are used here to exchange data between different configurations also called *context*. A data pipe contains a plurality of context switching logic arrays (CSLA) which can be used to process two 16-bit words. An incoming context, then, can pick its input data where its predecessor left off by acquiring the intermediate data deposited on the rightmost portion of the pipe and processing it in a pipeline from right to left. Unfortunately, the methods developed have remained in a conceptual stage. Neither time multiplexing nor context switching FPGAs have ever been commercialized.

4.3 Temporal placement

In 1996, Xilinx brought the famous FPGA 6000 [121, 37] series to the market. It was possible with those devices to carry only small modifications to the design, leaving the rest of the circuitry unchanged. Based on this architecture, some researchers made considerable progress in partial reconfiguration,[124, 121, 37]. After the disappearance of this device from the market, research in partial reconfiguration also slowed down. A couple of years ago, a new class of FPGA, the Virtex series which supports partial reconfiguration, has been brought to the market. Although this device has been used very successfully in different experiments, and despite the fact that it will remain on the market for a while, they have received little research or experimental focus. However, some authors have have managed to provide important contributions in partial reconfiguration [121, 52]. We present the work of Teich et al [121, 52] which reveals a strong formal basis for analysis and discussion.
In Teich et al [121, 52], a task is represented by a cube in which the x and the y coordinates represent the width and the height of the task in the given FPGA. The z coordinates represents the latency of the task. Given a precedence graph, the objective is either:

1. to finds the minimal execution time of the precedence graph on a fixed-size FPGA, or to
2. to finds the FPGA of minimal size to accomplish the tasks within a fixed limit of time

The authors modeled the problem as a three-dimensional **orthogonal packing problem (OPP)** which is the problem of deciding if a given set of boxes can be placed within a given container of size (h_x, h_y, h_t). The objective of finding the FPGA with minimal size to accomplish the given set of task in a time slot is reduced to the base **minimization problem (BMP)**, i.e the minimization

of h_x for a given h_t and a container with size (h_x, h_x, h_t) with quadratic base. The objective of minimizing the execution time of the graph given a fixed-size FPGA is reduced to the **strip packing problem (SPP)** which is the problem of minimizing h_t for a given base (h_x, h_y) such that all boxes fit in a container of size (h_x, h_y, h_t). The BMP and SPP problems are then optimally solved using the so called packing classes. A three-dimensional packing is first reduced to three one-dimensional packings. Then a set of **interval graphs** $G_i = (V, E_i)$ are constructed on the basis of the overlapping between components in the i-th dimension. An edge (u, v) belong to E_i, iff the projection of u and v overlap in the i-th direction. A powerful tool, the **packing class**, is then used to reduce the search space for a feasible packing. A packing class is defined as a 3-tuple of interval graphs satisfying the following conditions:

1. Any independent set of G_i is admissible, i.e all boxes in S must fit in the i-th dimension.

2. There must be at least one dimension in which the corresponding boxes do not overlap.

The search procedure is a branch and bound algorithm which works on the packing classes. To explain how it works the following terminology is used:

- An **induced cycle** in the graph $G = (V, E)$ is defined as a set of vertices $U = \{u_1, ..., u_k\} \subset V$, such that the cycle $C =< u_1, u_2, ..., u_k, u_1 >$ of edges is contained in E. The length of the cycle C in this case is k.

- An induced cycle is said to be **Chordless** if only the edges (u_i, u_{i+1}) are contained in E. If none of the edges (u_i, u_{i+2}) is contained in E, the induced cycle C is a **2-Chordless**.

The search tree is traversed by depth first search and the branching is done by fixing an edge $\{b, c\} \in E_i$ or $\{b, c\} \notin E_i$. After each branching step, it checks to see if one of two conditions (1 or 2) previously defined is violated or if a violation can be avoid by fixing further edges. The branch and bound method is used to eliminate a particular type of configuration in the search space. Those are the induced chordless cycles of length four in E_i, the 2-chordless odd cycles in the set of edges not belonging to E_i and the infeasible stable sets in E_i (Those for which an overlapping occurs in each of the three dimensions). Each time such a subgraph is detected in the search tree, the algorithm abandons the search on the corresponding node. With the temporal placement constraint, all the edges of the graph E_t are implicitly defined. Only the nodes of the graphs E_x and E_y have to be constructed. This simplifies the problem from a three-dimensional to a two-dimensional problem for which a solution can be found more efficiently.
The work of Teich et al is a great reference in temporal placement, due to the amount of theorems and the power of the model used. However, the approach has some limitations for our purpose.

1. The method has been modeled with the Xilinx 6000 architecture in mind. Since these devices have disappeared from the market, it is difficult to apply the results in practice. As we mentioned in section 3.2 we target Virtex devices in which reconfiguration is done by changing frames. Because a frame occupies a complete column, modifications need to be done in the model in order to tackle this problem. Because the height of the components is unlikely to be a complete column size, a grouping of components in column is better adapted for our target device.

2. The connection between the components, i.e. the communication between the nodes of the given DFGs does not play a role in this method. The method works on a pure packing basis aimed at place cubes not necessarily connected together in a container. Recall that our goal is to have connected components placed in the same area. This is not likely to happen with this method.

3. The method provided by Teich et al is to be applied at compile time on given precedence graphs. The method provides an optimal solution for such kind of "fixed problem" with the goals defined by the author. Meanwhile we would like to have a method able to deal with online problems, i.e. dynamic precedence graphs, even if the solution produced by the method is not optimal at all. In this work we provide a simple algorithm able to deal with dynamic task graphs. The goal is the computation of a minimum amount of packets to partially reconfigure the device at run-time. This can be done by grouping the components in clusters which will then be placed on the slots of an FPGA, previously divided in slots to accommodate one cluster.

4. The objective is either to find a FPGA with minimal size to accommodate a set of tasks given a fixed time or to minimize the computation time of a given function, with a fixed FPGA size. The first objective is not interesting for us, since the size of our device is fixed for a given problem. The second objective comes close to our goal, but we formulate our goal in terms of the volume of packets to be sent to device during the computation of a complete precedence graph.

Before explaining our work in the next chapter, we present two conceptual works done in online temporal placement.

4.4 Defragmentation and Relocation

So far we have considered only problems in which all the operators and their schedules are known at compile time. Online scheduling and placement of operators have been considered in [33, 37, 39, 38, 67, 68, 45]. The idea behind online scheduling is to plan the allocation of tasks not at compile time, but at run-time. Incoming modules are normally placed in a "ready to execute" list. Free spaces on the FPGA are allocated to the modules in the list on the basis of priorities previously defined. As modules are put on the device or removed from the device, holes are built on the surface. Ready tasks have to wait for enough free contiguous space before being scheduled for execution. This can lead to very long waiting time and affect the real time aspect of some algorithms. To solve this problem, some authors [33, 37, 39, 38, 67, 68, 45] have proposed the use of defragmentation and relocation. Their goal is to rearrange the task on the FPGA, in such a way that the holes become contiguous and provide enough space for the execution of more modules.
Diesel et al [37, 39, 38] extracted two sub problems which first have to be solved in order to satisfy the next allocation request to a dynamically reconfigurable FPGA. The first one is to identify a good allocation site for the incoming task and the second is to calculate a schedule for the compaction that:

1. frees the allocation site of other executing tasks as quickly as possible,

2. delays the tasks that are to be moved as little as possible and

3. completes the compaction of the tasks as quickly as possible.

A *visibility graph*, defined as the graph in which the nodes are the executing tasks is used to determine the cost of freeing the executing tasks from each candidate in $O(n)$ time. An edge (t_1, t_2) exists in the visibility graph if the node t_2 dominates node t_1, i.e t_1 and t_2 overlap in the row direction and t_2 is placed on a column greater than t_1. It can be determined in $O(n^3)$ time if an incoming task can be allocated with compaction. A *top cell interval* and a *right cell interval* for each executing task are defined when a request arrives. The **top cell interval of an executing task**

t is the set of possible locations where the bottom edge of the incoming task t_i will abut the top edge of t. The **top right interval of an executing task** t is similarly defined as the set of possible locations where the left edge of the incoming task t_i will abut the right edge of t. The set of cells at the intersection of the set of top and right cell intervals is a set B, which defines **the minimum cost locations for placing the incoming task** if it is to be allocated in the neighborhood of t. The visibility graph is built in time $O(n^2)$. The list of executing tasks is first sorted in column order. Vertex insertion is done in linear time by a depth-first search of vertices not visited before determining whether the task is to the right of the sub-graph or not. For each edge inserted, the distance from the parent to the newly added child is computed. After building the graph, each node is stored with the maximum distance the task can be moved to the right by summing the distance in a bottom-up fashion. The need to determine the compaction cost for the sites that cannot be freed of executing tasks is eliminated by the final step which takes $O(n)$ time. A more accurate search for the best site to place an incoming task, based on a genetic algorithm, is described in [39].

Compton et al [33] suggested the modification of existing FPGA architecture to support defragmentation and relocation. Their architecture, the *R/D-FPGA* is based on a partially reconfigurable FPGA. The column decoder, the multiplexer and the input tri-state drivers of the traditional FPGA have been replaced by a structure called a *staging area*. The staging area is a small SRAM buffer equal in size to one full row of programming bits. It acts as a buffer between the CPU and the FPGA. Once the information in the staging area is complete, the staging area is written in a single cycle at the row location indicated by the row address. A BUS architecture is used to permit module movement without affecting their I/O pins connections and causing a re-route of the signals. In order to reduced the complexity of the problem, single dimension of reconfiguration is targeted, i.e. modules can be relocated only in a given row or column (in this case it is a row). The chip row decoder has two additional registers and two inputs/one output multiplexer. This allows a vertical offset loaded in one or more registers, to be added to the incoming row address, resulting in the new relocated row address. One of the offset registers is a “read” register used during defragmentation for reading a relocated configuration off of the array and the other register is a “write” offset register, which holds the relocation offset used when writing a configuration. The original row address supplied to the reconfiguration hardware is simply the row address of that particular row within the reconfiguration. A column decoder between the staging area and the array is not necessary since the staging area is equal in width to the array and therefore each bit of the staging area is sent out on exactly one column.

The major problem of the R/D-FPGA architecture is the bus structure adopted to solve the I/O problem of moving components. While providing great flexibility, the arbitration of a bus connecting hundreds of modules becomes complex and slows down design.

The described work in temporal placement targets devices that permit a fine grained partial reconfiguration, i.e. any cell or CLB can be replaced without affecting their neighbors. Since the praxis is different, we believe that most of the related work helps us develop methods for specific devices, but can not be applied as presented by their authors. In practice, little work has been done in implementing partial reconfigurable systems. The partial reconfiguration capabilities of the Xillinx Virtex devices as well as the maturity of those devices have increased interest in using partial reconfiguration on practical cases. Systems allowing partial reconfiguration at run-time just started to appear [82, 66]. Examples of such are the implementation of a dynamic reconfigurable network packet processing application [82] and the implementation of a dynamic reconfigurable red Solomon decoder [66].

Chapter 5

Temporal Partitioning and Placement

In the first part of the chapter, we present three methods that we developed to compute the temporal partitioning of a given DFG. The first approach is an enhancement of the list scheduling method. A local optimization step is used to generate a minimum number of partitions through the use of the so called *configuration switching*. The second and completely novel approach uses a three dimensional *spectral placement* method to position the modules of the DFGs in a three dimensional vector space in such a way that the sum of the distance among the modules is minimized. A recursive bipartition approach is used to separate the partitions.
The second part of the chapter deals with two methods we developed to compute the temporal placement of the nodes of a DFG. The two methods are based on the placement of *clusters*[1] of components in a three dimensional vector space. The two approaches differ only in the computation of the clusters from a DFG. The first approach uses a level based assignment to cluster the components of the graph, while the second one relies on two dimensional spectral placement followed by a cluster growth method to build the clusters.

5.1 Temporal Partitioning

5.1.1 List Scheduling Based Approach

Traditional list scheduling based temporal partitioning algorithms can produce a series of configurations based on the same set of operators if the components are "well ordered"[2] in the list. For example, for two consecutive configurations $\zeta_i = \{C_1, ..., C_{k_i}\}$ and $\zeta_{i+1} = \{C_1\prime, ..., C_{k_{i+1}}\prime\}$ representing two partitions P_i and P_{i+1}, one can be the subset of the other one, i.e:

- $\zeta_i \subseteq \zeta_{i+1}$, or
- $\zeta_{i+1} \subseteq \zeta_i$

If one of those two situations arises, then the reconfiguration overhead can be reduced by implementing the two partitions P_i and P_{i+1} in one configuration $\zeta_{new} = \zeta_i \cup \zeta_{i+1}$. The components of ζ_{new} will the be shared among the two partitions P_i and P_{i+1}. That means, the modules required for both configurations are placed on the device and wired in two different ways. Each way corresponds to one configuration. Figure 5.1 shows a partitioning of a graph in two partitions P_0 and P_1. The set of components required to implement P_1 is a subset of the set of components required to implement P_0.

[1] A cluster defines a set of modules to be placed together on the FPGA.
[2] A well order defines the way components with the same level number should be place in the list before partitioning.

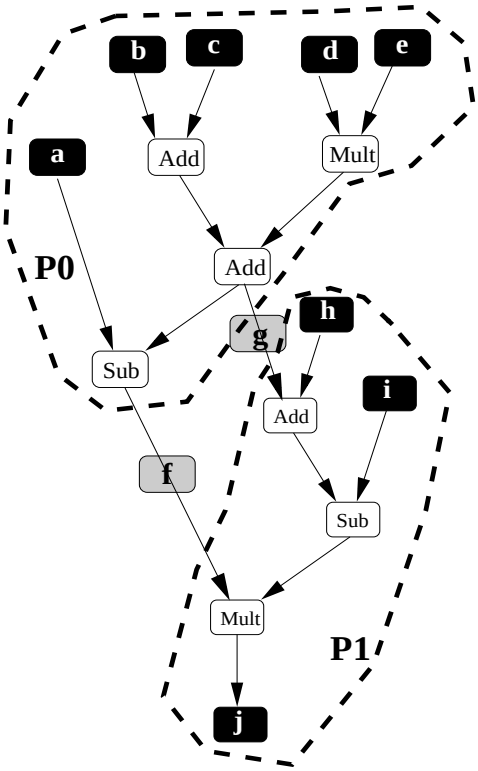
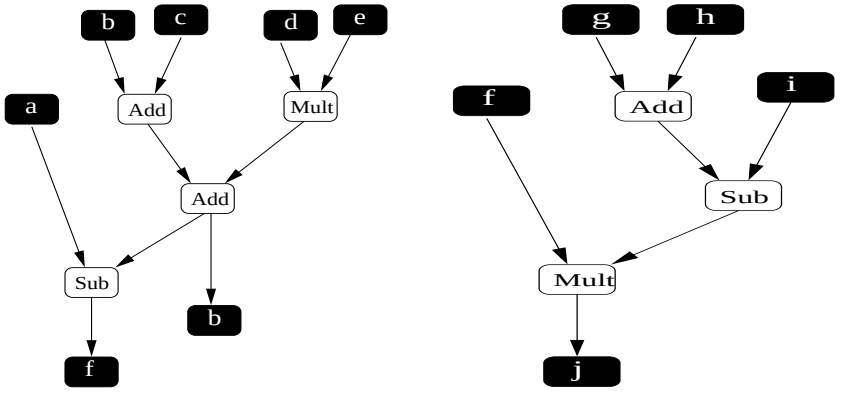


Figure 5.1: Partitioning of a graph in two set with a common set of operators

Without loss of generality, we will consider that none of the following conditions is met: $\zeta_i \subset \zeta_{i+1}$ or $\zeta_{i+1} \subset \zeta_i$. In this case the modules in $\zeta_i \cup \zeta_{i+1}$ build the set of operators to be implemented on the device. With the use of multiplexers on the inputs of the operators in $\zeta_i \cap \zeta_{i+1}$ and the use of selection signals to connect the corresponding signals to the module inputs, it is then possible to implement the connection defined in configuration ζ_i as well as those defined in ζ_{i+1} together. Switching from the configuration ζ_i to the configuration ζ_{i+1} can be done by setting the corresponding value of the selection signals. The device is therefore reconfigured without changing the physical configuration. We call this process *configuration switching*. With configuration switching there is no need to save the registers of the FPGA in the processor address space during reconfiguration because no register is altered. Therefore, configuration switching helps to reduce the data exchange between the processor and the FPGA.
For a series of configurations $\zeta_1, ..., \zeta_r$ we could extract the small amount of common operators needed to implement all the configurations and implement the configuration switching on $\zeta_1, ..., \zeta_r$. But the amount of multiplexers needed to assign the operators to a particular configuration as well as the difficulty to route all the possible configurations can jeopardize the implementation of configuration switching for the configurations $\zeta_1, ..., \zeta_r$. Therefore, we will limit the number of configurations to be implemented simultaneously on a device to two. This choice is probably not the optimal one, but it is the one which intuitively provides us with a greater chance of having many configurations being switched. The search for a tradeoff between the number of configurations to reside in the FPGA and the complexity of the final design is an optimization problem that we do not address in this work.
Figure 5.3 illustrates the implementation of configuration switching on a device for the two partitions of figure 5.2. During reconfiguration, the corresponding values for the selection signal are set on the multiplexers. The output of the corresponding partitions are then selected and sent to the operators. Although configuration switching can save time and reduce the data transfer be-



(a) $\zeta_1 = \{+, *, +, -\}$ (b) $\zeta_2 = \{+, -, *\}$

Figure 5.2: Example of two configurations ζ_1 and ζ_2 with $\zeta_1 \in \zeta_2$

tween the processor and the FPGA, it's implementation usually requires additional resources. If the additional resource becomes too great, then it makes no sense to have a smaller number of components implemented, and a larger logic area on the device just to realize the switch. The tradeoff between the number of additional resources and the number of configurations to be implemented has to be chosen carefully. If we take a close look at the way operators are implemented in FPGA[3], then we will find that many resources are left unused. Therefore, they can be used to implement the additional resources required for the multiplexers. Because the size of a multiplexer is the same as that of an adder, a subtractor or a register, one may be tempted to think that using one of those operators in configuration switching would double the operator size. This is not true. In fact adders, subtractors and registers require almost no additional resource in a FPGA to be implemented as shared components in configuration switching. If a register, for example, has to be shared in two configurations ζ_1 and ζ_2, then we need a condition $cond$ to assigned it either to ζ_1 or ζ_2. Each bit of the register is normally implemented in a half FPGA slice using one input of the corresponding LUT and the corresponding flip-flop. If I_0 is the input and O_0 the output of the register in the first configuration and I_1 its input and O_1 its output in the second configuration, then the logic required for configuration switching is defined via the following code segment: *if* $cond = 1$ *then* $(O_0 = O_1) = I_0$ *else* $(O_0 = O_1) = I_1$. The output is the same since only one resource is used. This code segment will be synthesized to produce the boolean equation $(O_0 = O_1) = I_0 * cond + I_1 * (not(cond))$ which can be implemented in a 3 inputs LUT. Since a physical register will need a minimum of a half slice of the FPGA to implement one bit, at least one 4-input LUT will be available meaning that we can implement configuration switching without additional physical resource. This is llustrated in figure 5.4, where the Input I_0 is attached to the F_1 input of the LUT in the half slice used by one bit of the register, I_1 is attached to F_1 and the outputs O_0 and O_1 are attached to the flip-flop output X_q. The switching condition $cond$ is attached to F_3. If $cond = 0$, then the register is available for the first configuration, otherwise it is available for the second configuration.
In many cases, the great part of the additional resources needed to implement configuration switching will be taken from the unused resources of the implemented operators. Because of the possibility of carrying out local optimizations during the partitioning process, we use list-scheduling and enhance it to implement configuration switching.

[3]We consider the operators available in the Xilinx Core Generator or in the JBits environment.

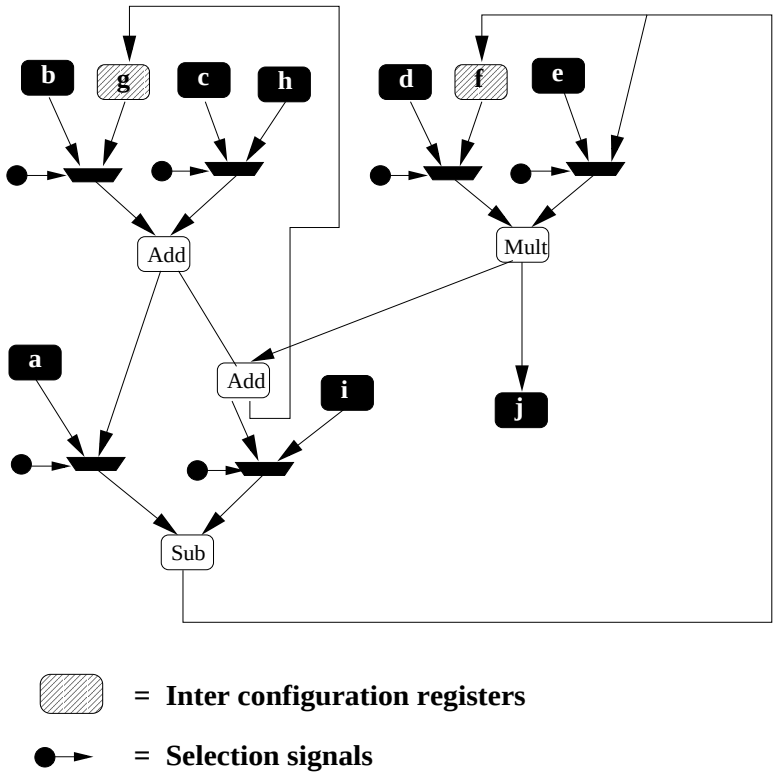


Figure 5.3: Implementation of configuration switching with the partitions of figure 5.2

The List Scheduling Based Algorithm

We now provide the description of our enhanced list scheduling based temporal partitioning algorithm. The method $generate\_partitions$ (Alg. 1) takes as inputs a DFG G and a device dev and returns a list of all partitions $list\_partitions$. All the nodes of the DFG are first inserted in $list\_nodes$ in order according to the precedence constraints of the DFG (line 2). The algorithm builds at each iteration step a pair of partitions ($first\_part, second\_part$) and tries to find out if it is possible with the set of operators in the two partitions to implement configuration switching. This is done by filling the partition $first\_part$ with the nodes of the DFG until the limit is reached (line 3 to 10). The second partition $second\_part$ is then built in the same way (line 11 to 18). The algorithm tests if the context switching can be implemented between these two partitions (line 19). This is done with the function $union\_fits\_on\_dev$ by checking if the union of their components

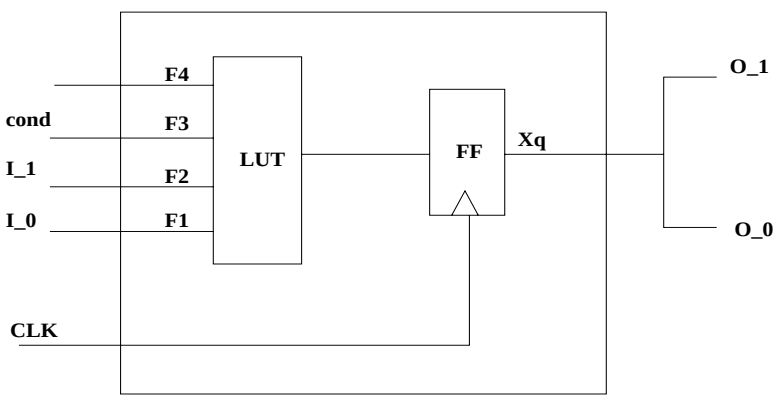


Figure 5.4: Sharing a register in two configurations without additional resource

Alg. 1 The LS-based temporal partitioning algorithm

```
1:  Algorithm generate_partitions(DFG G, DEVICE dev)
2:  list_nodes := generate_nodes_list_with_priority(G);
3:  first_part.reset();
4:  for (i := 1; i < list_nodes.size();i++) do
5:     choosed_node := get_next_node_without_pred(list_node);
6:     if ((first_part.size() + choosed_node.size()) < dev.size()) then
7:        first_part.insert(choosed_node);
8:     end if
9:  end for
10: list_partitions.insert(first_part);
11: second_part.reset();
12: for (i := 1; i < list_nodes.size();i++) do
13:    choosed_node := get_next_node_without_pred(list_node);
14:    if (second_part.size() + choose_node.size()) < dev.size()) then
15:       second_part.insert(choosed_node);
16:    end if
17: end for
18: list_partitions.insert(second_part);
19: if (union_fits_on_dev(first_part, second_part) ≤ dev.size()) then
20:    implement_config_switch(first_part, second_part);
21:    if (list_node.empty()) then
22:       GOTO 3:
23:    else
24:       Sop
25:    end if
26: else
27:    first_part = second_part
28:    if (list_node.empty()) then
29:       GOTO 11:
30:    else
31:       Sop
32:    end if
33: end if
```

will fit on the device. If so, then the function $implement\_config\_switch$ is called on line 20 to realize this. Multiplexers are added on the inputs of the common operators and the inputs for the two partitions are wired to the inputs of the multiplexers. The outputs of the common operators are connected on one side to the inputs of the operators as defined in the first partition and on the other side to the inputs of the operators as described in the second partition (figure 5.3). A new pair of partitions is generated and the process repeats. If context switching cannot be implemented, then the second partition $second\_part$ becomes the first one $first\_part$ and a new second partition is filled with the nodes from the DFG (line 26 to 32). The algorithm stops when the list of nodes $list\_nodes$ is empty.

The positions of the components in $list\_nodes$ play a great role in this algorithm. Normally, the components are placed in the list $list\_nodes$ in increasing DFG level number order. For maximum resource sharing between the partitions $first\_part$ and $second\_part$, components of the same type and with the same precedence level should not be placed consecutively in the list. For example, if we have four multipliers and four adders with the same level number to be placed on the list, then we should not place all the multipliers before placing the adders. If we do this, then the list-scheduling method will first remove all the multipliers in front of the list and place them in the first partition before placing the adders in the second partition. This leaves no way to implement configuration switching. To avoid this, the positions of the operators should be mixed in the list. A reasonable ordering places one multiplier, then one adder, then one multiplier and then one adder and so on. In this way, the two partitions $first\_part$ and $second\_part$ will contain both multipliers and adders. This allows us to implement configuration switching on these two partitions more efficiently. The function $generate\_nodes\_list\_with\_priority$ orders the components in the list $list\_nodes$ as described here.

Example: Numerical Solution of a Differential Equation

To illustrate and show the efficiency of the method described in this section, a numerical method for solving a differential equation as described in [121] is considered. Fig 5.5 shows the DFG for solving a differential equation of the form $y'' + 3xy' + 3y = 0$ in the interval $[x_0, a]$ with step size d_x and initial values $y(x_0) = y_0$, $y'(x_0) = u_0$, using Euler's method. The VHDL specification of the equation solver is given in Alg. 2 and the corresponding DFG in figure 5.5. We consider

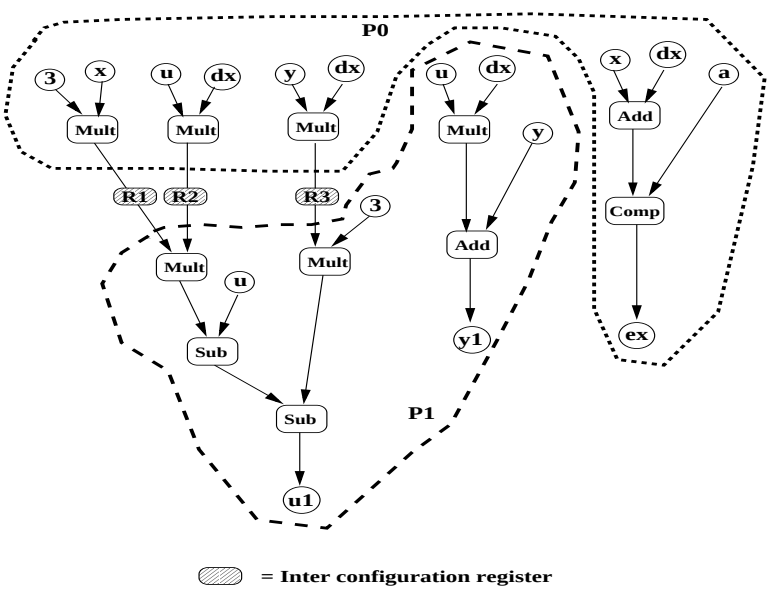


Figure 5.5: DFG For the DE-Integrator

Alg. 2 VHDL specification of the differential equation algorithm

```
1: ENTITY DGL IS
2: port(x_in, y_in, u_in, dx_in, a_in): IN REAL;
3: act:IN Bit; y_out: OUT REAL;
4: END DGL;
5: ARCITECTURE Behave of DGL is;
6: Begin
7: Process(act)
8: Variable x, y, u, dx, a, x_1, y_1, u_1: REAL;
9: Begin
10: x := x_in; y := y_in; u := u_in;
11: dx := dx_in; a := a_in;
12: LOOP
13: x_1 := x + dx;
14: u_1 := u - 3*x*u*dx - 3*y*dx;
15: y_1 := y + u*dx;
16: x := x_1; y := y_1; u := u_1;
17: EXIT WHEN x_1 > a;
18: END LOOP
19: y_out <= y;
20: END PROCESS;
21: END Behave;
```

the worst case, where multiplexers have to be added for each operator. The size (in CLBs) of the different modules for the Xilinx Virtex architecture is given in table 5.1. As we can see, the

Cores	X_Size	Y_Size
16-Bit Adder	8	1
16-Bit Subtractor	8	1
16-Bit Comparator	8	1
2 In/1-Out 16-Bit MUX	10	1
16-Bit Multiplier	5	20

Table 5.1: Cores Size (in CLB) for the Xilinx Virtex Architecture

size of a 2-inputs/ 1-output 16-Bits-Multiplexer is (8×1). Since the size of a 16 Bit-multiplier is (5×20), sharing a multiplier module in two configurations will halve the size of the multiplier (5×10) and double the size of the multiplexer $(2 \times (8 \times 1))$ on the resulting configuration. The net gain is about 80 CLBs. Given the number of multipliers in the design, configuration switching can be expected to perform well. When targeting the Virtex FPGA family above the Virtex 300 with a minimum area of (32×48), the design will completely fit in. It requires only one partition and reconfiguration is not needed. But, if we target the smaller Virtex 100 with size (20×30), we must limit the configuration to three simultaneous multipliers in order to preserve enough space for routing. Without configuration switching, a temporal partitioning algorithm produces a minimum of two configurations in order to implement the DFG (figure 5.5).

For a thousand iterations of the DFG computations, the device must be configured a thousand times. FPGA reconfiguration time is typically measured in milliseconds. The resulting overhead incurred for reconfiguration alone is measured in the thousands of milliseconds. Configuration switching obviates the need for physical reconfiguration. We simply switch from one configura-

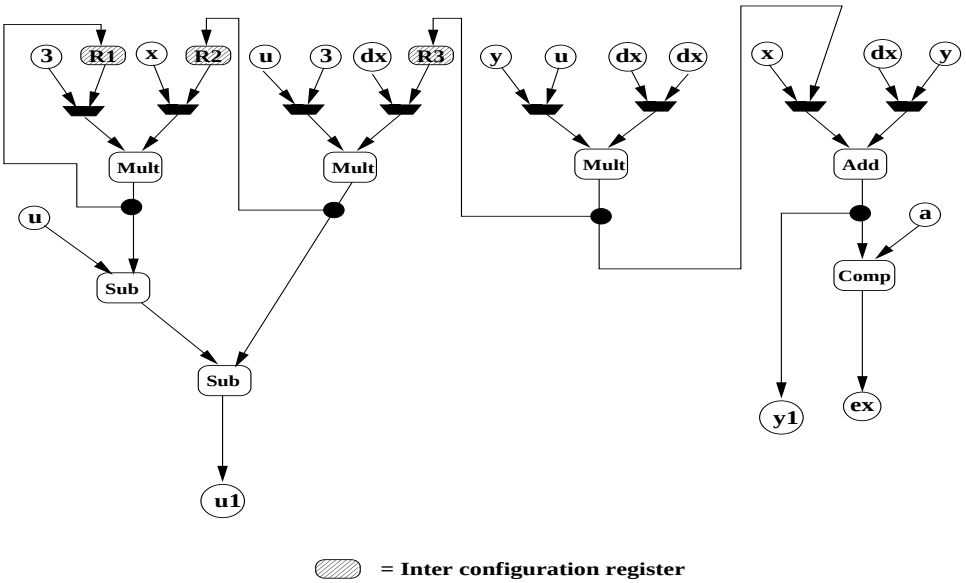


Figure 5.6: Implementation of configuration switching for the differential equation integrator DFG

tion to the next by setting the corresponding value of the selection signals. This has the effect of reducing reconfiguration time, and driving the level of data exchange to zero. The device will contain three multipliers, two subtractors, one adder, one comparator and three registers to temporally hold the computation result on reconfiguration. In the worst case[4], 8 additional multiplexers will be required. The total number of CLBs needed in this case is 420, i.e the design easily fits in the target device (Virtex 100). It occupies about 70% of the device size. The resulting configuration is given in figure 5.6.
The example we have presented here shows that the number of physical configurations can be halved by our enhancement. This situation will arise if two consecutive partitions share the same set of components. Our enhancement is therefore important when partitioning a graph using the list scheduling method.

5.1.2 Spectral Methods

In chapter 3 we defined our goal in temporal partitioning as the minimization of data exchange between the processor and the FPGA during the computation of a DFG. The minimization of data exchange between the processor and the FPGA also means minimization of communication between partitions, since the communication between the partitions is done via the processor which temporally holds the content of the interconfiguration registers during the reconfiguration. We saw that this goal could be reached by placing connected components in the same partition. For this purpose we defined the connectivity of a graph as the function to be optimized. The connectivity of a graph can be minimized by placing components in an n dimensional space in such a way that the sum of the distance between component pairs is minimized. This approach is called the wire length model. Since the sum of the distances between component pairs can be minimized if connected components are placed in close proximity, the wire length model is likely to provide an optimal placement of the components in an n dimensional space. Our problem can be solved using the wire length if the following two sub problems are solved:

[4]Worst case means, that the CLBs occupied by the operators do not have free resources to implement the multiplexing logic.

1. Placement of the components in an n dimensional vector space to minimize the sum of the distance between the components.

2. Derivation of a partition from an optimal placement which minimizes the sum of the distance between the components.

We start with the first problem by considering the one dimensional[5] version as defined by Hall in [69] and illustrated in figure 5.7. (The subfigure (a) shows an example of a directed graph with 4 nodes and 3 edges, while subfigure (b) shows the one-dimensional placement on a line. The small black squares represent the centers of the logic cells. Subfigure (c) shows the two-dimensional placement of the same nodes. The eigenvalue method takes no account of the logic cell sizes or actual location of logic cell connectors. The scaling of the computed position (by the width and height of the node bounding box) must be done. In subfigure (d) a complete layout is made by placing the logic cells on valid locations, leaving room for the routing). Given a DFG $G = (V, E)$,

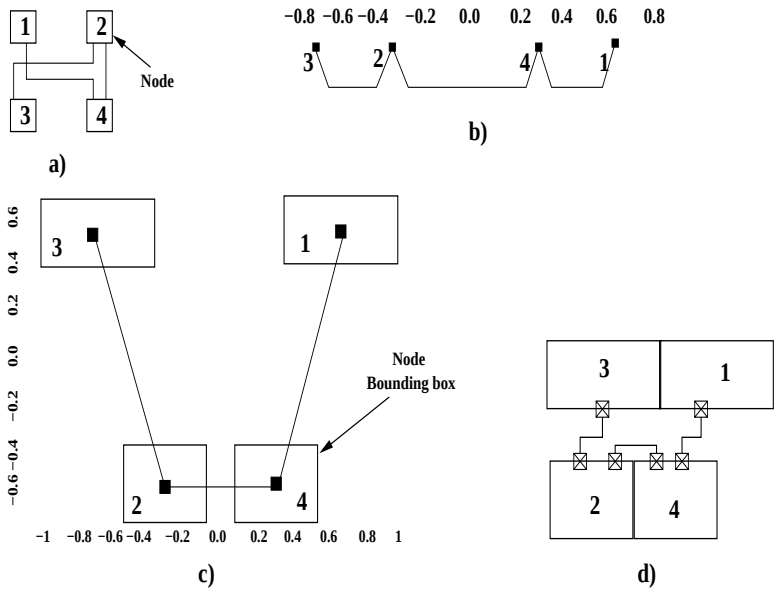


Figure 5.7: 1-D and 2-D spectral-based placement of a graph.

find locations for the $|V|$ nodes which minimize the weighted sum of squared-distances between the nodes. If x_i denotes the X-coordinates of node $v_i \in V$ and r denotes the weighted sum of squared distances between the nodes, then the 1-dimensional problem is to find the row vector $X^T = (x_1, x_2, ..., x_n)$ which minimizes:

$$r = \sum_{i=n}^{n} \sum_{j=n}^{n} (x_i - x_j)^2 w_{ij} \tag{5.1}$$

To avoid the trivial case in which $x_i = 0$ for all i, we impose the following condition (normalization):

$$X^T X = 1 \tag{5.2}$$

We assume that the non interesting solution $x_i = x_j$ (for all $i, j \in \{1, .., n\}$) is to be avoided.
We first define the connection matrix, the degree matrix and the Laplacian or disconnection matrix of G as follows:

[5]Placement on a line

Definition 15 (Connection Matrix, Degree Matrix, Laplacian Matrix) *Gievn a DFG $G = (V, E)$, we define:*

- *The connection matrix of G as the symmetric matrix $C = (c_{i,j})$ with $(1 \leq i, j \leq \mid V \mid)$ and $c_{i,j} = 1$ if $(v_i, v_j) \in E$ and $c_{i,j} = 0$ otherwise.*
- *The degree matrix of G as the diagonal matrix $D = (d_{i,j})$, $(1 \leq i, j \leq \mid V \mid)$ with $i \neq j \rightarrow d_{i,j} = 0$ and $d_{i,i} = \sum_{j=1}^{\mid V \mid} c_{i,j}$.*
- *The Laplacian matrix of G as the matrix $B = D - C$.*

For two nodes v_i and v_j of the DFG connected by an edge the matrix the connection matrix will have an entry one in line i and column j. The degree matrix is a diagonal matrix. An entry in the diagonal (line i, column i) correspond to the number of nodes adjacent to v_i. The Laplacian matrix is simple the difference between the degree and the connection. Hall has proved in [69] that:

$$r = X^T B X \tag{5.3}$$

Since B is positive semi-definite ($B \geq 0$) and B is of rank $|V| - 1$, whenever G is connected [69], the initial problem is now reduced to the following:

$$\begin{cases} \text{minimize } r = X^T B X \text{ with } B \geq 0 \\ \text{subject to } X^T X = 1 \end{cases} \tag{5.4}$$

This is a standard constraint optimization problem which can be solved using the method of the Lagrange multipliers. This is a standard method used to find the extrema of a function $f(x_1, ..., x_n)$ subject to a constraint $g(x_1, ..., x_n) = 0$. An extrema exists if equations (5.5) and (5.6) are satisfied.

$$df = \frac{\partial f}{\partial x_1} dx_1 + \cdots + \frac{\partial f}{\partial x_n} dx_n = 0 \tag{5.5}$$

$$dg = \frac{\partial g}{\partial x_1} dx_1 + \cdots + \frac{\partial g}{\partial x_n} dx_n = 0 \tag{5.6}$$

Now if we multiply (5.6) by a parameter λ to be determined and subtract the result from (5.5), we obtain equation (5.7):

$$(\frac{\partial f}{\partial x_1} - \lambda \frac{\partial g}{\partial x_1}) dx_1 + \cdots + (\frac{\partial f}{\partial x_n} - \lambda \frac{\partial g}{\partial x_n}) dx_n = 0 \tag{5.7}$$

Because the differentials are all independent, we can set any combination equal to zero and the remainder must still give zero. This requires:

$$(\frac{\partial f}{\partial x_k} - \lambda \frac{\partial g}{\partial x_k}) dx_k = 0 \; \forall k \in 1, ..., n \tag{5.8}$$

The constant λ to be computed is called the **Lagrange multiplier**.
To solve problem 5.4, we apply the method of the Lagrange multipliers with $f = X^T B X$ and $g = X^T X - 1$. We introduce the Lagrange multiplier λ and form the Lagrangian $L = X^T B X - \lambda(X^T X - 1)$ as shown in equation 5.7. Taking the first partial derivative of L with respect to X and setting the result equal to zero yields to equation 5.9

$$2BX - 2\lambda X = 0 \iff (B - \lambda I)X = 0 \tag{5.9}$$

Multiplying equation (5.9) by X^T and applying the constraint (5.2), equation (5.9) yields a non trivial solution if and ondly if X is the Eigenvector of B which minimizes r and $\lambda\,(=r)$ is the corresponding Eigenvalue. Because the minimum Eigenvalue $\lambda_0 = 0$ yields the non interesting solution $X^T = (1, 1, ..., 1)/\sqrt{(n)}$, the second smallest Eigenvalue λ_1 should be chosen. The Eigenvector X_1 related to the Eigenvalue λ_1 is the solution to the one dimensional problem (5.4).
For a placement in a k-dimensional vector space, the problem is formulated similar to the one dimensional version. The entire dimensions involved have to be considered:

$$\begin{cases} \text{minimize } R = {X_1}^T B X_1 + {X_2}^T B X_2 + ... + {X_k}^T B X_k \\ \text{subject to } {X_1}^T X_1 = {X_2}^T X_2 = ... = {X_k}^T X_k = 1 \end{cases} \tag{5.10}$$

(X_i defines the coordinates of the nodes of V in the $i-th$ dimension) has to be solved. Analog to the 1-dimentional case, the Lagrange multiplier method will be applied with the r (each for one dimension) Lagrange multipliers $\lambda_1, \lambda_2, ..., \lambda_k$. The solution are the Eigenvectors associated to the r smallest non zero Eigenvalues $\lambda_1, \lambda_2, ..., \lambda_k$. This approach is known in the literature as **spectral method**. Spectral methods have been widely used in the past for partitioning and placement [8, 9, 79, 29, 41, 40]. It's run-time is dominated by the computation of the eigenvalues for which can be done using various methods on different architectures. The most used algorithm for computing the eigenvalues of a matrix is the Golub-Kahan [58] method. It needs $O(n^3)$ on a single processor to compute the eigenvalues of a n by n matrix. Using $O(n)$ processors, the eigenvalues can be compute with a parallel version of the Hestenes method [122, 109, 81, 22] in $O(n^2 S)$ where S is the number of so called sweeps [81, 22]. Brent and Luk [22] conjectured that $S = log(n)$ and therefore $S \leq 10$ in general. For sparse and quadratic matrices, the eigenvalues can be computed in $O(n^{1.4})$ using a more efficient Lanczos method [58, 8].

5.1.3 Application to the Temporal Partitioning Problem

So far we have shown that the spectral method can be use to solve the first part of our problem. The second part, the generation of partitions from a placement with minimum wire length, has yet been considered.
We consider the placement of the components in a three dimensional space in which the X axis and Y axis represents the FPGA surface and the Z axis represent the time at which each component should be mapped inside the FPGA. With the spectral approach we are able to generate a placement with minimum wire length. In the time dimension, the precedence constraints between the components of the DFG define the following additional constraints for the problem (5.10).

$$\forall e = (v_i, v_j) \in E, z_i \leq z_j \tag{5.11}$$

For large graphs, the number of constraints which can be formulated in (5.11) can be too high. Therefore, solving (5.10) with the additional constraints (5.11) becomes too complex for large graphs. We overcome this difficulty by solving problem (5.10) in a 3 dimensional vector space without taking the constraints (5.11) into consideration. Then we select the nodes for each partition using an iterative approach.
Once a 3 dimensional spectral placement of the component is computed, we need to build the partitions. We do this by incrementally generating the partitions $P_0, P_1, ..., P_k$ using recursive bisection of the sets $\tilde{P}_i$, which are the set of the modules to be partitioned. Initially we set $\tilde{P}_0 = V$. At step i, the Partition $\tilde{P}_i$ is divided into two partitions P_i and $\tilde{P}_{i+1}$. This process is repeated unstill $\tilde{P}_{i+1} = \emptyset$. The partition P_i is built at step i by picking components along the Z-axis and placing them in P_i until the size of P_i reaches a given limit (the device size). This process creates

a bipartition $(P_i, \tilde{P}_i - P_i = \tilde{P}_{i+1})$ of the set of components not yet assigned to a partition. For each computed bipartition $(P_i, \tilde{P}_{i+1})$, we seek exactly one of the following situations:

1. $(P_i \leq \tilde{P}_{i+1})$
2. $(\tilde{P}_{i+1} \leq P_i)$
3. There is no relation between P_i and $\tilde{P}_{i+1}$

This means that either no edge exists that connects a component of one partition with a component of another partition, or all the edges connecting components between the two partitions have the same direction. This is not always true, since we have used an undirected graph for the placement, and we did not consider the precedence constraints of (5.11). The bisection has to be improved to fit the cycle free constraint. We do this by moving nodes from one side of the bisection to the other until a strict order is reached between P_i and $\tilde{P}_{i+1}$.

Example

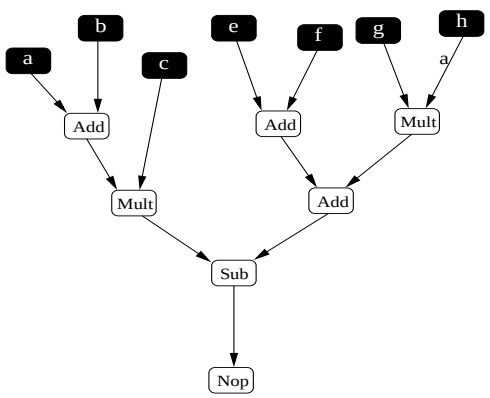


Figure 5.8: DFG for the computation of ((a+b)*c) - ((e+f)+(g*h))

To illustrate our temporal partitioning using the spectral method, we consider the graph of figure 5.8. The following matrix represents the laplacian matrix D of the given graph as described in section 5.1.2. Using this, we computed the three smallest Eigenvalues and the three Eigenvectors related to them. The results are given in figure 5.9. Using those three Eigenvectors we computed the 3-D placement of figure 5.10. By picking the component along the Z-axis in increasing order of their Z-coordinates, we constructed the two partitions P_0 and P_1 of figure 5.11. This partition is not ordered since the edges $(3, 6)$ and $(4, 6)$ have their source in P_0 and their destination in P_1, while the edge $(5, 3)$ has it's source in P_1 and it's destination in P_0. This produces a configuration graph with a cycle. By exchanging the nodes 3 and 5 from P_0 to P_1 and vice versa, the partitioning becomes an ordered one. Our spectral method for temporal partitioning is self explained and described in algorithm 3.

5.1.4 Elimination of Cycles in the Configuration Graph

As stated earlier, the resulting bisections are not always cycle free, since we are working on an undirected graph model. In the previous example we could solve the problem easily by manually exchanging two nodes. This is not possible for large graphs. Therefore we would like to compute the lines 7 to 9 of algorithm 3 automatically. To do this we rely on a well known tool: the *iterative*

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & -1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 3 & 0 & -1 & -1 & 0 & 0 \\
-1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 3 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & -1 & 0 & 3 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 3 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 3 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 1
\end{pmatrix}$$

Table 5.2: Laplacian matrix of the graph of figure 5.8

Alg. 3 The spectral-based temporal partitioning algorithm

```
1: Place all the components in a 3-D using a spectral method
2: while not (all component have been allocated to a partition) do
3:    Initialize a new partition
4:    while (not exceed device size) do
5:       Select components by increasing order along the Z-axis and place them in the partition
6:    end while
7:    while (the bipartition is non ordered) do
8:       Exchange the nodes connected by the crossing edges until an ordered bisection is pro-
         duced.
9:    end while
10: end while
```

improvement procedure of Kernighan and Lin (KL) described in [93, 53, 98]. Since the KL-algorithm deals only with undirected graphs and because we target directed graphs in this work, some modifications have to be done on the original KL algorithm.

The fundamental idea behind the KL-algorithm is the definition of a *cut* of a bisection as well as the notion of the *gain* of moving a vertex from one side of the bisection to the other. For an undirected graph, a cut is defined as the weighted sum of all the edges crossing from one partition to another. By moving a node from one partition into the other, the number of crossing edges is also modified and the value of the cut changed. The KL-algorithm allows a series of moves which reduce the bisection cut. If the gain of moving a vertex is positive, then making that move will reduce the total cost of the cut in the partition. During one iteration of the KL-algorithm, nodes are moved from one side of the bisection and locked on the other side. The cost of swapping unlocked nodes in opposite parts is then computed and the nodes with the best gain (greatest decrease or less increase of the cut) are swapped. If all the nodes are locked, the lowest cost partition is set to the current computed partition, if it improves the cost of the cut. One iteration of the KL-algorithms is called a *pass*. After one pass, all the nodes are unlocked and a new pass is computed. The iteration terminates if a pass produces no further improvement on the cut. For a more detailed description of the KL-methods and it's extension by Fiduccia and Mattheyses, refer to [93, 53, 98].

3 smallest non zero Eigenvalues of the disconnection (laplacian) matrix

$$\lambda_6 = 0.112586$$
$$\lambda_3 = 0.267949$$
$$\lambda_1 = 0.438447$$

(a) 3 smallest Eigenvalues of the graph of figure 5.8

Subspace related eigenvectors

X-axis	Y-Axis	Time-Axis
0.0410593	0.0	0.34188
-0.152761	0.0	0.0749482
0.225048	0.0	0.191984
-0.241072	0.325058	-0.0749481
-0.241072	-0.325058	-0.0749483
0.355147	0.0	-0.191984

(b) Corresponding Eigenvectors

Figure 5.9: 3 smallest Eigenvalues and the related Eigenvectors of the laplacian matrix

Since the graphs targeted in the original approach are undirected, it doesn't matter if an edge crosses from the first partition to the second partition or vice versa. In our case, directed graphs are targeted and the original algorithm has to be modified to fit our needs. To better explain how the modification is done, we first provide more definitions.

Definition 16 *We now consider the original directed graph $G = (V, E)$. For two nodes v_i and v_j in V and a bipartition P, Q of G, we have:*

- $E_P(v_i) = \sum_{(v_i,v_j)|v_j \in P} w_{ij}$ *is the weighted sum of the edges from v_i to P, i.e edges connecting v_i to nodes in P. (figure 5.12)*
- $I_P(v_i) = \sum_{(v_j,v_i)|v_j \in P} w_{ji}$ *is the weighted sum of the edges from P to v_i, i.e edges connecting nodes in P to v_i(figure 5.12) and*
- *Let $E_{P,Q} = \{(v_i, v_j) \in E \mid v_i \in P$ and $v_j \in Q\}$ is the set of edges crossing from partition P to partition Q, i.e. edges having their sources in P and their destination in Q.*

Recall that at step i of the temporal partitioning (Alg. 3), a bipartition $(P_i, \tilde{P}_{i+1})$ is generated. Our goal is to have either $E_{P_i,\tilde{P}_{i+1}} = \emptyset$ or $E_{\tilde{P}_{i+1},P_i} = \emptyset$. Intuitively we would like to combine the two

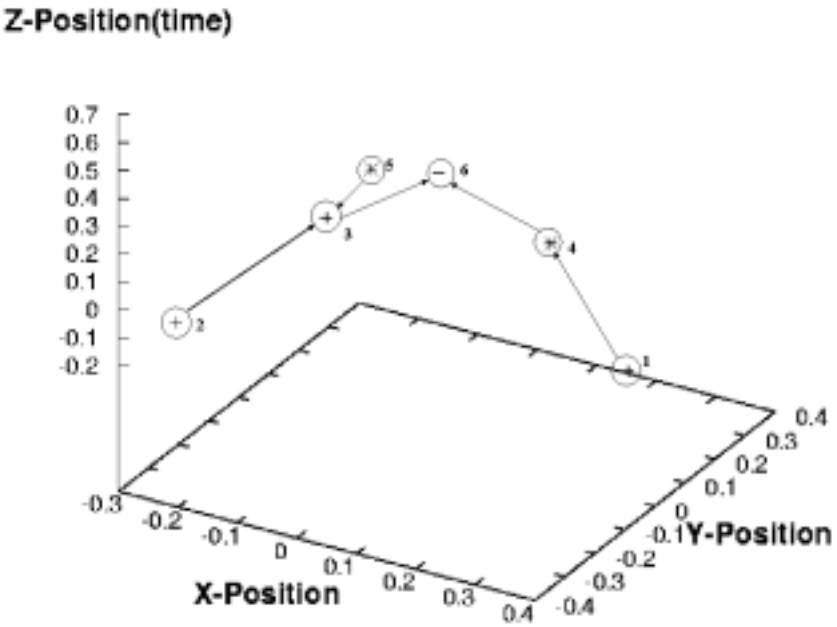


Figure 5.10: 3-D spectral placement of the DFG of figure 5.8

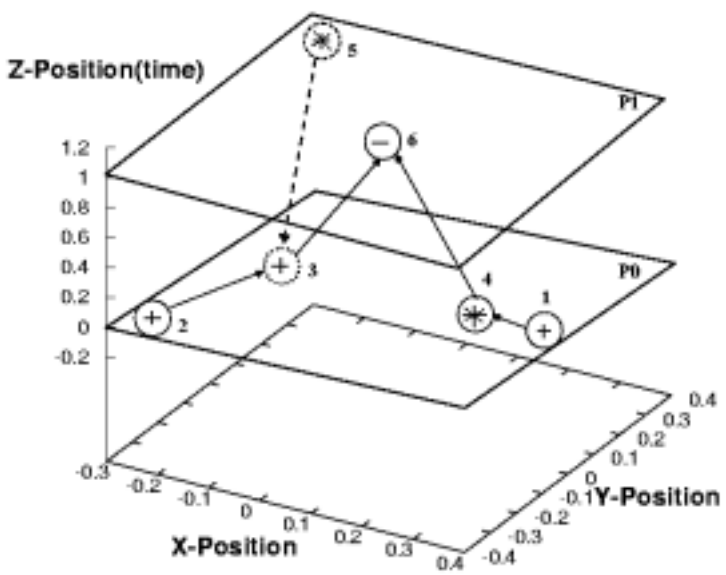


Figure 5.11: Derived partitioning from the spectral placement of figure 5.10

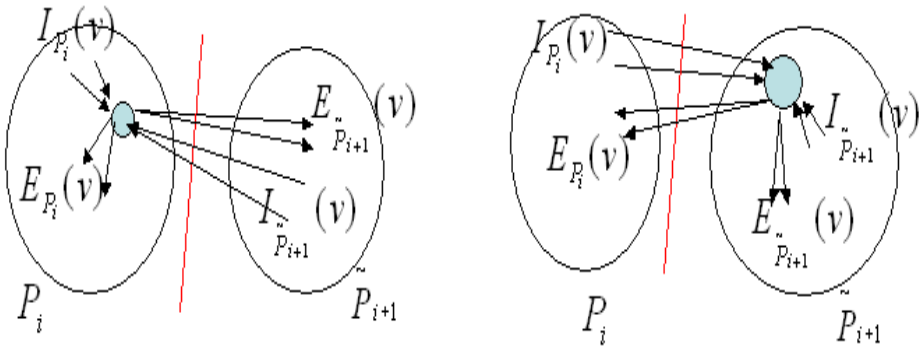


Figure 5.12: Internal and external edges of a given nodes

variables $E_{P_i,\tilde{P}_{i+1}}$ and $E_{\tilde{P}_{i+1},P_i}$ in the definition of our cut and try to decrease one of them to zero using the KL-algorithm. But decreasing one variable could have a negative effect of increasing the second one, thus producing a cycle of improvement and alteration on the cost of the cut. To avoid this we apply two instances of the KL-algorithm on the same bisection in parallel. The objective is to have $E_{P_i,\tilde{P}_{i+1}} = \emptyset$ on the first computation path and $E_{\tilde{P}_{i+1},P_i} = \emptyset$ on the second one. Obviously the cut is set to $|E_{P_i,\tilde{P}_{i+1}}|$ on the first path and $|E_{\tilde{P}_{i+1},P_i}|$ on the second path. After one pass on each path, we check if the objective has been reached on one path. If this is the case, then the result is set to be the partition generated on that path. Otherwise, a new pass is computed on the two computation paths (see Alg. 4). The gain of moving a node is defined differently on the two

Alg. 4 The modified KL-algorithm for the nodes arrangement

```
1: Initialize two computation paths
2: while (|E_{P_i,P̃_{i+1}}| ≠ 0 and |E_{P̃_{i+1},P_i}| ≠ 0) do
3:     Compute a new KL-pass on the first path
4:     Compute a new KL-pass on the second path
5: end while
6: if (|E_{P_i,P̃_{i+1}}| = 0) then
7:     result of partition = result of first path
8: end if
9: if (|E_{P̃_{i+1},P_i}| = 0) then
10:    result of partition = result of second path
11: end if
```

computation path.

- On the first path where the goal is to have $|E_{P_i,\tilde{P}_{i+1}}| = 0$, the gain of moving a node j from P_i to $\tilde{P}_{i+1}$ is $I_{P_i}(v_j) - E_{\tilde{P}_{i+1}}(v_j)$ and the one of moving a node k from $\tilde{P}_{i+1}$ to P_i is $E_{\tilde{P}_{i+1}}(v_k) - I_{P_i}(v_k)$.

- On the second path where to goal is to have $|E_{\tilde{P}_{i+1},P_i}| = 0$, the gain of moving a node j from P_i to $\tilde{P}_{i+1}$ is $E_{P_i}(v_j) - I_{\tilde{P}_{i+1}}(v_j)$ while the gain of moving a node k from $\tilde{P}_{i+1}$ to P_i is $I_{\tilde{P}_{i+1}}(v_k) - E_{P_i}(v_k)$.

The gain defined on each computation path is the same like the one defined in the original KL-algorithm. The modified version of the KL-algorithm presented here will produce the desired result on one path. Because the targeted graphs are acyclic DFGs, there exists a partition in which all the edges cross from the first one to the second (such a partition is provided for example by a list-scheduling algorithm). The cost of the cut is 0 in this case. By decreasing the value of the cut of the initial partition on both computation paths, the pair wise interchange algorithm of Kernighan and Lin will compute on one path a partition with edges having the same direction.
Applying our method to the partition of figure 5.11, a pair wise interchange of the nodes 3 and 5 generates the desired result. The algorithm finds this solution in only one pass.

5.2 Temporal Placement

If the device has partial reconfiguration capabilities, then partial reconfiguration will help reduce the data exchange between the processor and the FPGA. As stated in section 3.2 reconfiguration is made by exchanging frames. We seek the sequence of configurations which minimizes data exchange between the processor and the device. This goal can be reached if modules which should be reconfigured at the same time are placed in consecutive frames. This requires a viable clustering strategy on the given DFG. Our strategy is a two-step method based on:

1. The computation of the clusters of components to be placed at the same time on the device, and

2. The temporal placement of the computed clusters on the device.

While the second step can be computed by a fast and easy procedure, step one requires a more careful clustering strategy. Therefore we first explain how a set of clusters produced in step one is temporally placed before explaining how the clusters are extracted from a DGF.
A cluster is a set of components to be placed at the same time on the device. To be able to free the space occupied by the components of a cluster, those components should have approximately the same run-time. We first provide some definitions that will be used in the rest of the chapter.

Definition 17 (Clustering) *Given a DFG $G = (V, E)$,*

- *a clustering $C = \{C_1, ..., C_n\}$ is a partition of $G = (V, E)$ into the disjoint subsets C_1,...,C_n.*
- *The run-time $t(C)$ of a cluster C is the maximal run-time over all it's components, i.e. $t(C) = max\{t_i | v_i \in C\}$.*

Definition 18 (Cluster Graph) *Given a DFG $G = (V, E)$ and a clustering $C = \{C_1, ..., C_n\}$ of G, the cluster graph $Cl(G)$ of G is the graph in which the nodes are the clusters. An edge exists between two nodes C_i and C_j of $Cl(G)$ if an edge exists in E which connects a component of C_i with a component of C_j.*

Similar to the temporal partitioning problem (def. 7) in which the goal is the computation and scheduling of a configuration graph (def. 9), our goal in temporal placement is the computation and temporal placement of a cluster graph.
Given DFG $G = (V, E)$ and a clustering $C = \{C_1, ..., C_n\}$ which is a partition of $G = (V, E)$, we can temporal placed the clusters $C_1, ..., C_n$ in a first-fit fashion. For each cluster C_{act} ready[6] to be placed on the device, the first space which can hold C_{act} is chosen to place C_{act}. That means,

[6] If all the predecessors of a cluster in the cluster graph are placed, then the cluster is said to be ready.

the cluster C_{top} with the minimum run-time among the clusters already placed in the FPGA is selected and C_{act} is placed on top C_{top}, if doing so will not lead to an overlapping between C_{act} and other some clusters already placed on the device. The method is similar to the first-fit memory allocation method. If the clusters have different sizes, then the first-fit method is likely to produce holes during temporal placement. To avoid this situation, we require the same size for all the clusters. It will therefore be possible to place any ready cluster on top of the cluster with the earliest finishing-time.

Given a DFG $G = (V, E)$ and a clustering $C = \{C_1, ..., C_n\}$ of G where all the clusters have the same size, algorithm Alg. 5 computes a temporal placement of C in a first-fit manner. For a new cluster C_{act} to be placed on the device, the algorithm checks for the cluster C_{top} with the minimum run-time among the clusters allocated to the device. The cluster C_{act} is placed on top of C_{top}. Placing C_{act} on top C_{top} simply means that the finish time of C_{top} is the starting time of C_{act}. The algorithm stops when all the clusters have been placed. Figure 5.13 illustrates the temporal

Alg. 5 A simple procedure to temporal place clusters on an FPGA

1: **while** All the clusters are not placed **do**
2: Select the next cluster C_{act} to be placed
3: From the clusters already placed, select the one C_{top} with the smallest run-time.
4: place the cluster C_{act} on top of C_{top}
5: **end while**

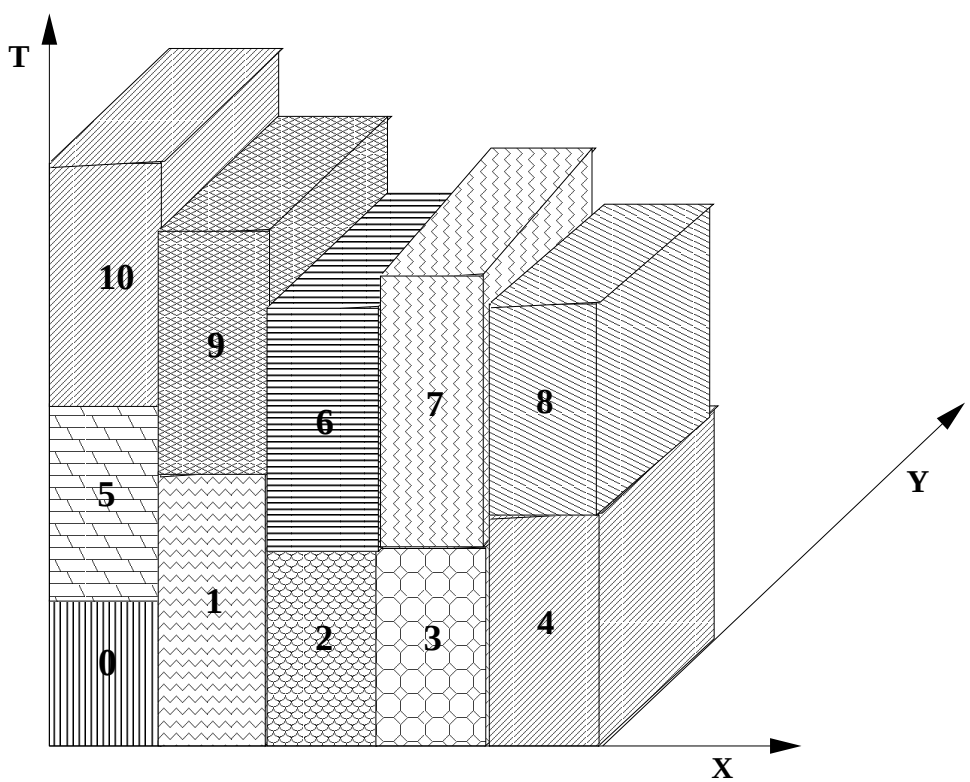


Figure 5.13: Temporal placement of a set of clusters

placement of a set of ten clusters. At initialization the clusters C_0, C_1, C_2, C_3 to C_4 are completely placed on the device. In order to place cluster C_5, cluster C_0 with the earliest finish time is selected and cluster C_5 is placed on top of C_0. Cluster C_6 occupies the space freed by cluster C_2 and cluster C_7 occupies the space freed by C_3. C_8 is then placed on top of C_4, C_9 on top of C_1 and C_10 on

top of C_5. The configuration path produced by this example is $(\{0,1,2,3,4\} \rightarrow \{5,1,2,3,4\} \rightarrow \{5,1,6,7,4\} \rightarrow \{5,1,6,7,8\} \rightarrow \{5,9,6,7,8\} \rightarrow \{10,9,6,7,8\})$. If we assume that each cluster consists of three columns or frames, then the cost of this computation is $12+3+6+1+1+1 = 24$ frames.
The method we presented here is a heuristic for which we cannot prove the efficiency. But, the fact that components which can be downloaded and replaced at the same time are grouped in clusters before being "temporally" placed is not only likely to reduce the total volume of packets needed during the computation of a given function, but guarantees that no component will be disturbed during the partial reconfiguration of the device. The method we provided is similar to the memory allocation method in traditional general purpose computer. It can also be used for online temporal placement of tasks on a partial reconfigurable FPGA.
We now present the two methods we developed for computing the clusters from a given a DFG. The first approach is a "level-based" clustering and the second one is spectral clustering based on the computation of the Eigenvalues of the Laplacian matrix of the given graph.

5.2.1 Level-based clustering

The level-based approach is similar to the list-scheduling temporal partitioning. Because the algorithm is based on the computation of a level number, we will first provide the definition of a level number before explaining how the algorithm works.

Definition 19 (Level Number) *For a given node v_i in the DFG $G = (V, E)$, we define the level number $level(v_i)$ of v_i (figure 5.14):*

- $level(v_i) = 1$*, if v_i has no predecessor, and*
- $level(v_i) = (max\{level(v_{i1}), ..., level(v_{ik})\} + 1)$ *if $v_{i1}, ..., v_{ik}$ are the predecessors of v_i.*

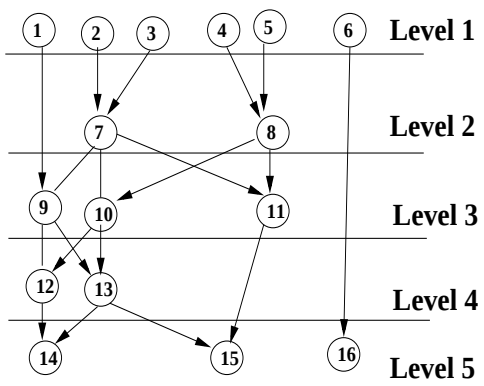


Figure 5.14: Level assignment

Our level-based clustering method (Alg. 6) first assigns a level number to all the components of the given DFG. The nodes with the same level-number are then assigned to a common a **group** labeled with the common level-number of its components. The clusters are now built from the groups by putting components with the same run-time together, until the limit on the size of the clusters is reached.
In this form, the algorithm will put components whose run-time is different from that of all other components in their own clusters, thus wasting device space. To avoid this, we will place components with approximately the same run-time in the same clusters. Anomalous components that

Alg. 6 The level-based clustering algorithm

```
1: Assign a level number to each component of the graph
2: Place the components of the graph in groups according to their level-number and assign the
   common level number of the components to the groups they belong to.
3: for each group G do
4:    initialize a new cluster C
5:    select a new component v_i from G and place it in C
6:    while (size(C < limit)) do
7:       select a component v_j aus G
8:       if t_i = t_j then
9:          remove v_j from G and place it in C
10:      end if
11:   end while
12: end for
```

do not group are merged into larger clusters. In level-based clustering, the two main criteria for belonging to a cluster are the level number and the run-time. The algorithm can be applied as well for run-time clustering. Incoming nodes will be assigned a level number and put in the available groups. The run-time of the incoming components will then determine in which cluster the components should be placed. Since all the components are not arriving at the same time, a time period for checking the available components should be set. The algorithm will then run periodically to allow new components to be considered during the clustering.
Because the interconnections between the components in the DFG are not the decisive factor during the clustering, this method is suitable for DFGs with few interconnections among the modules. If the graph has a lot of interconnections between the nodes, then the interconnections among the components should intuitively govern the clustering rather than the level number of the components. When clustering the graph of figure 5.14 using the level-based approach, the nodes 1, 2, 3, 4, 5 and 6 will be first grouped together and then placed in the same cluster depending on their run-time. Intuitively, we prefer to compute a more efficient clustering in which components 2, 3 and 8 belong to a first cluster, components 4,5 and 7 to the second cluster, components 6 and 16 in a third clusters and so on. As stated in 5.2.3, the construction of the spectral method presented in this chapter insures that connected components are placed nearby each other before partitioning. Therefore, the spectral method will place the tightly connected components 2, 3 and 7 together in one cluster, and components 4,5 and 8 will appear in the second cluster.

5.2.2 Spectral Based Clustering

The objective of a clustering algorithm is the computation of compacts and well separated clusters. That means a clustering in which two connected components appear in the same cluster while two unconnected components appear in different clusters. If we define the **diameter of a set C** as:

$$diam(C) = \max_{u,v \in C}(dist(u,v)) \tag{5.12}$$

and the **split of two sets** C_1 **and** C_2 as:

$$split(C_1, C_2) = \min_{u \in C_1, v \in C_2}(dist(u,v)) \tag{5.13}$$

Where $dist(u,v)$ is the Euclidian distance between two components u and v, then we can say that clusters with small diameter are compacted while clusters with large splits are well separated. In

order to compute compacted and well separated clusters, we define three possible objectives for a clustering algorithm. For a clustering $C = \{C_1, C_2, ..., C_k\}$, we define:

- **Max-Min-Split**: Maximize

$$f(C) = \min_{1 \leq i < j \leq k} (split(C_i, C_j)) \tag{5.14}$$

- **Min-Max-diameter**: Minimize

$$f(C) = \max_{1 \leq i \leq k} (diam(C_i)) \tag{5.15}$$

- **Min-Sum-diameters**: Minimize

$$f(C) = \sum_{i=1}^{k} (diam(C_i)) \tag{5.16}$$

Objective (5.14) is used to compute a well separated set of clusters, while objectives (5.15) or (5.16) seek the computation of compact clusters. We learn from [8] that objective (5.14) can be optimally solved in polynomial time for any k and any dimension, while objective (5.15) and (5.16) are NP-complete for any $k > 2$ and any dimension $d > 1$.
Our strategy consists of:

1. using the spectral method to compute a compact cluster. This will help to fulfill objectives (5.15) or (5.16).

2. using an optimal method to separate the clusters and fulfill objective (5.14)

The computation of compact clusters in step one is done by placing the components of the graph in a two dimensional vector space in such a way as to minimize the sum of the distances among all the components. The objective function defined in equation 5.1 ensures that the connected modules are placed in the same area by a spectral method, thus leading to the fulfillment of objectives (5.15) and (5.16). Apart from the fact that components of a cluster should be connected, we prefer to have components with approximately the same run-time belonging to the same cluster. The connectivity factor is captured in the connection matrix used for the spectral placement while the time is not. We therefore propose a formula to capture the time and connectivity factors together inside the connection matrix. An entry c_{ij} in the connection matrix is defined in equation (5.17).

$$c_{ij} = \begin{cases} \alpha w_{ij} + \beta \text{ if } t_i = t_j \\ \alpha w_{ij} + \beta \frac{1}{|t_i - t_j|} \text{ otherwise} \end{cases} \qquad \text{with } \alpha + \beta = 1 \tag{5.17}$$

The factor α captures the importance of the connectivity of two components, while the factor β captures the importance of the run-time difference between two components. If the run-time difference between two components is large, then those two components should not be placed in the same cluster. This explains the fraction based on the run-time difference of components.
For the separation of clusters and therefore the fulfillment of objective (5.14), we use a so called **cluster-linkage or cluster grow** method which optimally solve objective (5.14) [8]. The starting point of the algorithm is the 2-D placement computed by the spectral method. Having the 2-D placement, the method begins with each component in its own cluster. The clusters are then successively merged to form a larger one. At each step of the merging process, a pair of clusters with minimum distance is merged together. The algorithm terminates when a given limit[7] is reached.

Alg. 7 The spectral-based clustering algorithm

```
1: Compute a 2-D spectral placement
2: Set the limit criterion for the merging of clusters
3: for each component i in the graph do
4:    Create a cluster C_i and assign i to C_i.
5:    Put C_i in the list L of clusters to be merged
6: end for
7: while L not empty do
8:    Select a pair of clusters (C_i, C_j) with minimum distance
9:    if (C_i, C_j) fulfill the limit criterion then
10:      merge C_i and C_j in one new cluster C_k and put C_k in L
11:      remove C_i and C_j from L
12:   end if
13: end while
```

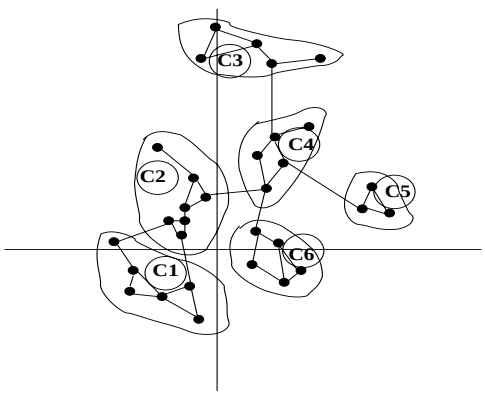


Figure 5.15: A spectral-based clustering

Algorithm Alg. 7 computes a clustering with the spectral method for a given DFG G and a limit on the size of the clusters. Figure 5.15 shows a clustering example with clusters $(C_1, C_2, ..., C_6)$ built from a 2-D placement of the nodes of a graph. A limit of 7 nodes[8] is set on the number of nodes in a cluster.
The spectral clustering method presented in this section has the advantage of directly providing the relative position of the components in their clusters. Further, the compaction of components in clusters is guaranteed by the 2-D spectral placement. In order to ensure a good separation of clusters, a minimum distance can be imposed for each pair of clusters. In practice we found that it was difficult to define a formula for a minimum distance. We prefer a limit on the size of the clusters which is helpful for temporal placement.

5.2.3 Selection and Evaluation

So far we have presented two different approaches for computing a temporal partitioning and two different approaches for computing the temporal placement of a given DFG. Having a problem instance, we should be able:

1. to choose the best method for the temporal partitioning or temporal placement, and

[7]The limit on the size of the cluster
[8]It's assumed that all the nodes have the same size

2. to evaluate the results of the partitioning or placement produced by the method used.

In chapter 3, we defined the connectivity of a graph as the instrument to measure how strong the components of the graphs are interconnected. In section 4.1.1, one limitation of the list-scheduling and the level-based clustering methods has been shown to be the assignment of modules to partitions or clusters primarily on the basis of their level number, rather on than their interconnectivity. The construction of the spectral method presented in this chapter ensures that connected components are placed proximate to one another before partitioning or clustering. A connected group of components is therefore likely to be placed in the same partition or cluster. In this case we can apply list-scheduling or level-based on a graph only if its connectivity is very low, i.e. the components of the graph are not tightly connected. Under these conditions, the quality of list-scheduling and level-based approaches will not differ too much from that of the spectral approach. Therefore, it is more reasonable to choose the list-scheduling for temporal partitioning and the level-based method for temporal placement since they normally perform faster than the spectral method. If the graph is tightly connected, then we will have high connectivity. The quality of list scheduling for temporal partitioning, or level-based techniques for temporal placement, are low relative to the spectral method.
For a given DFG and a limit on the connectivity, the list-scheduling or level-based methods are best if the connectivity of the graph is below the given connectivity threshold, otherwise the spectral method works best.

Chapter 6

Results

In this chapter we evaluate the temporal partitioning and temporal placement methods which were developed and presented in chapter 5. To do this, we randomly generated a set of dataflow graphs, let the algorithms run on them and measured the performance of the algorithms according to the quality criteria introduced in definition 8. We introduced a second evaluation criteria that we call the **wasted resource (wr)** of a cluster and show how it's influenced by the weighting of the time factor in the spectral method. Due to the lack of a universal benchmark for algorithms working on dataflow graphs, we generated a set of fifteen graphs with various numbers of nodes and edges. A technique commonly used is replication where a basic graph is built and replicated to generate a larger one. Edges and nodes can be added or removed in order to increase or decrease the connectivity of the graph. The basic graphs which were replicated and modified are the dataflow graphs for computing the singular value decomposition [19] and the dataflow graph for computing the Fourier summation [48]. The results of the computation on the benchmark as well as the comparison of the different methods are presented below. We do not provide a comparison of our implementation with other methods for two reasons.

1. As we explained in chapter 4 our goal in temporal partitioning and temporal placement is different from those of many authors working in the same field.

2. The lack of a universal basis for evaluation of dataflow graph algorithms makes it difficult to compare the different methods.

Our temporal partitioning and temporal placement methods were evaluated separately on the basis of the benchmark.

6.1 Temporal Partitioning

Table 6.1 shows the result of the temporal partitioning with the list-scheduling method and the spectral method. For each graph, we listed the number of nodes, the number of edges, the connectivity of the graph (Definition 3), the quality of partitioning with list-scheduling and the quality of partitioning with the spectral method (Definition 8). Evident in this benchmark (table 6.1) is that the connectivity of a graph becomes smaller as the graph grows. This is due to the fact that, for a graph $G = (V, E)$, the connectivity is computed on the basis of number $(|V|^2 - |V|)$ of edges in the corresponding complete graph. This number grows quadratically with the number of nodes. In our benchmark, the dataflow graph's nodes are operators like adder multipliers, dividers, constant multipliers, constrained to two data inputs. Operator output can be sent to many nodes. This has the potential to increase the number of adjacent edges to that node, making it difficult to build very

Graph Name	#(Nodes)	#(Edges)	Connectivity	Quality SP	Quality LS
svd_complex6	173	257	0.0172738	0.0211765	0.0160372
svd_complex7	174	288	0.0191349	0.0824392	0.0813421
svd_complex3	115	158	0.0241037	0.0500857	0.0380134
svd_complex2	115	174	0.0265446	0.0163934	0.00633176
svd_complex4	107	171	0.0301534	0.018	0.00595943
svd_complex1	122	224	0.0303482	0.0163934	0.00898308
Fourier2	80	97	0.0306962	0.128689	0.1181489
svd_complex5	91	136	0.0332112	0.115052	0.101334
Fourier1	77	100	0.0341763	0.0976741	0.0876887
medium3	122	293	0.0396965	0.025	0.0160667
Fourier4	84	167	0.0479059	0.138539	0.121685
Fourier3	83	171	0.0502498	0.113115	0.100857
medium2	73	154	0.0585997	0.0277778	0.00757663
medium4	79	186	0.06037	0.025	0.011365
medium1	80	205	0.0648734	0.0305556	0.00989011

Table 6.1: Benchmark for the temporal partitioning methods

dense networks using two input - one output nodes. Therefore, the ratio of the number of existing edges over the number of "possible" edges in the graph will be low for large graphs. The re-

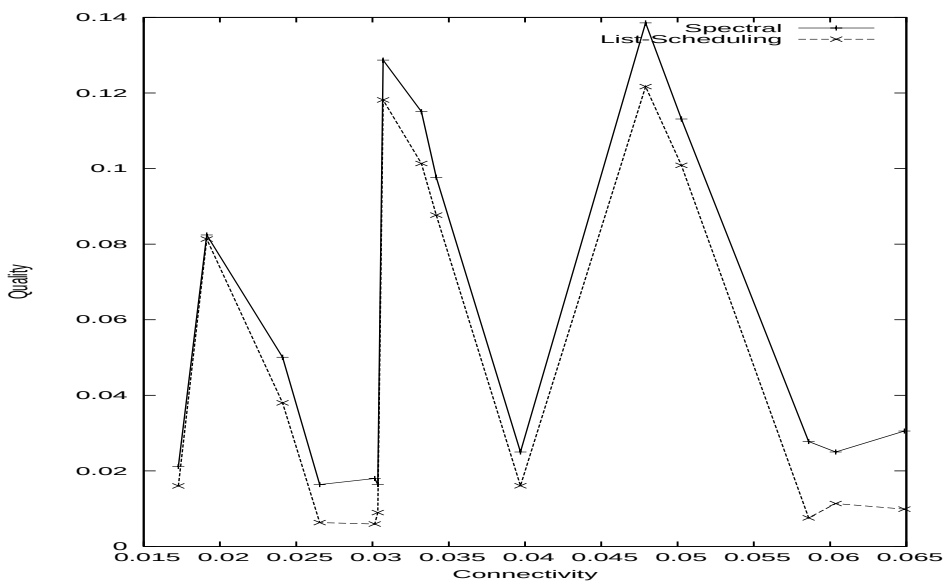


Figure 6.1: Result of the LS and spectral partitioning on the benchmark

sults provided in table 6.1 and illustrated in figure 6.1 confirm the assumption we made in section 5.2.3, i.e. that spectral partitioning should be preferred over list-scheduling for highly connected graphs. The graph shows the quality of the list-scheduling method and the spectral method as a function of graph connectivity. As we can see, for graphs with low connectivity, the quality of partitioning with list-scheduling is close to the quality of the partitioning with the spectral method (the two curves are very close at the beginning). The quality of the spectral method is better, but the difference with list-scheduling is too small. Because of its run-time, list-scheduling should be used in this case. As the connectivity of the graphs grows, the spectral method becomes better and the difference of quality for the two methods also grows. This is better illustrated in figure 6.2 which displays the quality difference between the spectral method and the list-scheduling method as function of the graph connectivity. The curve is very irregular, but it shows a general growth in

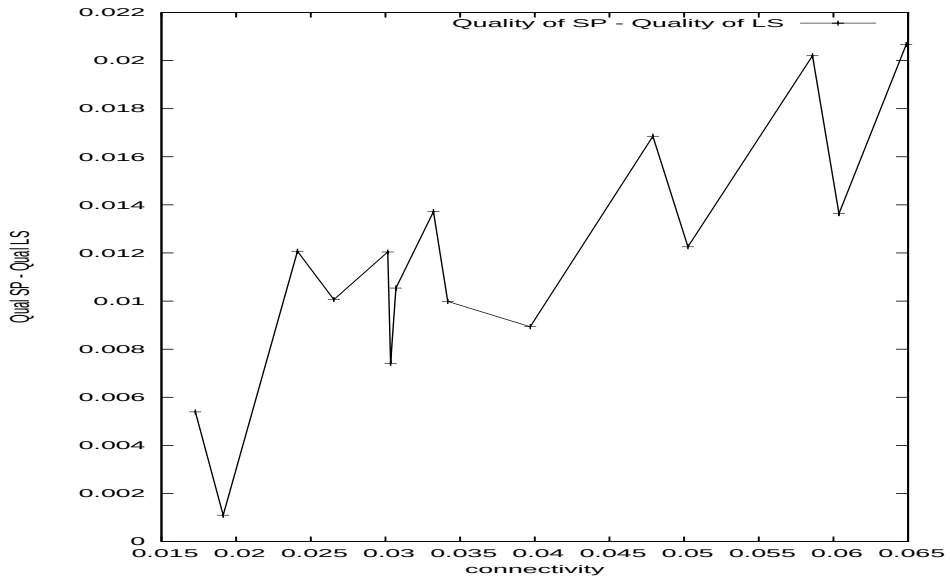


Figure 6.2: Difference of quality between the spectral and the list scheduling method

the quality difference between the spectral method and list-scheduling.
Figure 6.1 also shows an irregularity in the evolution of the quality curves for the two methods. We can explain this irregularity only by providing the quality of a partition as a function of the connectivity. This was not the goal sought by the benchmark. We made the assumption that the spectral method performs better than list-scheduling for highly connected graphs. The result of the benchmark shows that the spectral method is in general better than list-scheduling and therefore confirmed our assumption.
For large graphs which are generally low connected, list-scheduling will be preferred over the spectral method because the spectral method provides a narrow improvement in connectivity for a higher run-time. As the size of the graph decreases, the connectivity increases and the spectral method delivers better performance.

6.2 Temporal Placement

As in the case of temporal partitioning, we will not present the results of the comparison between our methods and other approaches. To our knowledge, the only work which dealt with this problem is the work of Teich et al [121, 52] that was presented in section 4.3. They used an optimal algorithm for a 3-D packing of the nodes of a dataflow such as to minimize the FPGA area of the run-time. Contrary to these objectives, our goal is neither the computation of the FPGA with minimal size to compute a set of tasks, nor the computation of the minimal run-time for a set of tasks given a fixed-size FPGA. The objective of computing a minimal size FPGA for a given set of tasks is not interesting for us, since the size of our device is fixed. The objective of minimizing the run-time of all the tasks for a given FPGA comes closer to our goal, but we focused on the minimization of communication as well as the reconfiguration overhead which was not considered in [121, 52]. We formulated our objectives in section 3.2 and the importance of clustering was proven through figure 7.3 to be high. Therefore, the benchmark will be used to highlight the advantages of each of the methods we have developed and explain the influence of weighting connectivity by the factor α and time by the factor β in equation 5.17. Recall that this equation

was defined in section 5.2.2 as follow:

$$c_{ij} = \begin{cases} \alpha w_{ij} + \beta \text{ if } t_i = t_j \\ \alpha w_{ij} + \beta \frac{1}{|t_i - t_j|} \text{ otherwise} \end{cases} \qquad \text{with } \alpha + \beta = 1 \tag{6.1}$$

with c_{ij} being an entry in the connection matrix used to compute the eigenvalues used for the spectral method.
The superiority of the spectral method relative to list-scheduling in temporal partitioning has been proven in the previous section. This superiority is expressed in term of the quality of the algorithm for a given connectivity. We have shown that the spectral method out performs list-scheduling for graphs with high connectivity. This result can be transfered to temporal placement because the clustering of a graph in temporal placement is nothing else than graph partitioning, except that the size of the partitions (clusters in this case) are just a portion of the FPGA size. Because the level-based clustering works on the same basis as list-scheduling partitioning and the spectral partitioning works on the same basis like the spectral clustering, the results (superiority of the spectral method toward list-scheduling for highly connected graphs) obtained in the previous section hold for this section.
We will now investigate the influence of weighting the connectivity by the factor α and time by the factor β in Equation 6.1 on spectral method clustering. As explained in the previous chapter, robust clustering groups connected components in the same cluster. Because all the components of a cluster are replaced at the same time, the run-time difference of the components belonging to the same cluster should not be too high. Otherwise, components with shorter run-times will remain idle for a longer period of time, resulting in a waste of FPGA resources. Equation 6.1 was developed to capture the time factor as well as the connectivity factor in the connection matrix. The goal was not only to group connected components in the same clusters, but also components with approximately the same run-time. To compare the level of wasted resources for each of the two methods, we first have to define what a wasted resource of a partitioning is.

Definition 20 (Wasted Resource) *Given a DFG $G = (V, E)$ and a cluster $C_i = \{v_{i_1}, .., v_{i_n}\}$, we define the wasted resource $wr(C_i)$ of C_i (figure 6.3) as the unused FPGA area occupied by C_i during it's computation. Recall that $t(C_i) = max\{t_j | v_j \in C_i\}$ was defined as the run-time of cluster C_i. With this, the wasted resource can be formally defined as follow:*

$$wr(C_i) = \sum_{j=1}^{n} (t(C_i) - t_j) \times a_j$$

The wasted resource of a node v_j in a cluster is the idle time of v_j in the cluster multiply by its area. The wasted resource of a cluster is the sum of the wasted resource of its components. Given a DFG $G = (V, E)$ and a clustering $C = \{C_1, ..., C_k\}$ of G, we define the wasted resource of C as the sum of the wasted resource of all it's cluster.

$$wr(C) = \sum_{i=1}^{k} wr(C_i)$$

The investigation of the influence of equation 6.1 on clustering with the spectral method has been done using different values for the pair (α, β). For each pair of values, we computed the quality difference between the spectral and level-based methods and the wasted resource difference between the two methods. The level-based method first assigns components of the same level to a common group. Inside a group, components are placed in clusters on the basis of their run-time. Therefore the level-based method is likely to waste fewer resources than the spectral method if the time factor is not considered in the connection matrix. If the time-factor is considered in the connection matrix (by the use of equation 6.1), then the spectral method wastes fewer resources. These assumptions should be supported by our benchmark. We ran the benchmark for the follow-

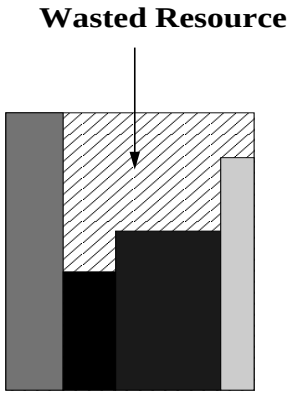


Figure 6.3: The wasted resource of a cluster.

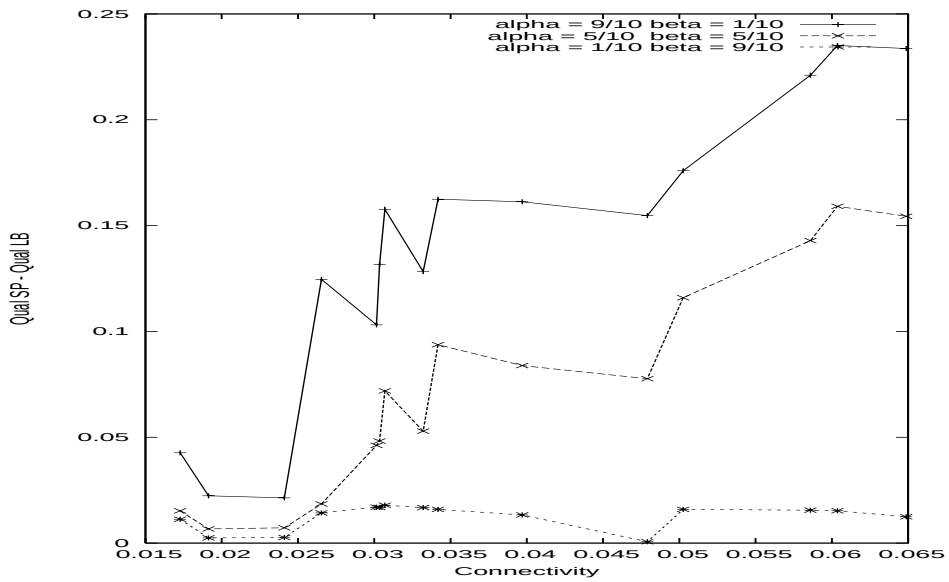


Figure 6.4: Influence of equation 6.1 on the quality difference between between the spectral and the level based clustering methods

ing pair of values: $(\alpha = 9/10, \beta = 1/10)$, $(\alpha = 5/10, \beta = 5/10)$ and $(\alpha = 1/10, \beta = 9/10)$. In other words, we weighted the connectivity with a factor of 9/10 and the time with a factor of 1/10 in the first case. In the second case we weighted the time and connectivity with the same value and in the third case, the time was weighted with higher value (9/10) than the connectivity (1/10). As expected, the quality difference between the spectral and the level-based methods decreases as the weight of the connectivity decreases. This is illustrated in figure 6.4. The figure shows that the curve comes closer to the x-axis while the connectivity weight α decreases. In the last curves, where the connectivity weight is only 1/10, is too close to the x-axis, i.e. the difference on the quality of the spectral method an the level-based method is close to zero.

As the weight of connectivity α decreases and the weight of time β increases, the wasted resource difference between the spectral and the level-based methods also decreases, meaning that the run-time becomes important in the spectral clustering method. This is illustrated in figure 6.5 in which the curve comes closer to the x-axis as the time weight β grows.

The last curves for which the time is highly weighted (9/10) relative to the connectivity (1/10) shows a very narrow wasted resource difference between the spectral and the level-based methods. We conclude this chapter by saying that the benchmark has provided strong support for the analysis of the method developed in the previous chapter. Further, the argument we made for choosing one

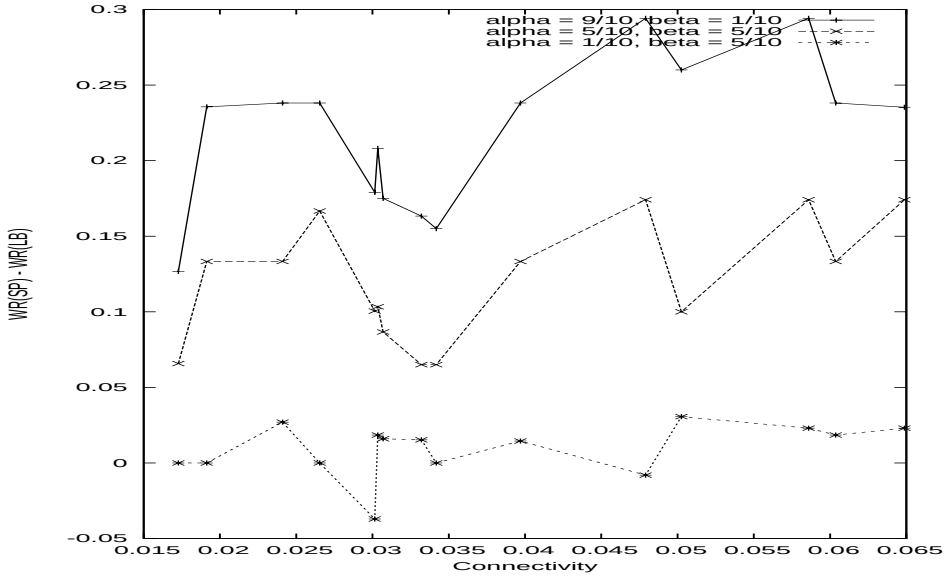


Figure 6.5: Influence of equation 6.1 on the wasted resources difference between the spectral and the level-based clustering methods

method over the other has been substantiated. The methods we have developed are implemented in the CoreMap design environment that we present in the next chapter.

Chapter 7

The CoreMap Design Environment

This chapter presents our design tool called CoreMap which was developed to ease implementation on a reconfigurable device. The CoreMap provides many features for fast and easy implementation and modification of FPGA designs.
With common design tools it is difficult to carry out modifications on a design without running the entire tool on the design chain. A fast modification or implementation is therefore difficult. To be able to modify the design quickly, a geometrical control of the modules and their interconnection by the designers is required. This functionality is not available in most of the commercial FPGA design tools. Some vendors (like Xilinx) provide tool functionality to assign a portion of the FPGA to a computing block. But the effort that a designer has to spend for implementation is high. To overcome these difficulties, we follow a core based approach. IP (intellectual Property) cores, that we simply call cores, are developed and provided via internet and other channels by companies specializing in particular functional implementation for FPGAs. They are designed by teams of experts, who have a high degree of understanding of the FPGA's structure, and who make their best efforts to provide the user with efficient blocks. These blocks can be combined together to generate algorithms which can be configured at run-time, and directly mapped to the FPGA. Using a core based implementation allows designers operate at a higher level of design abstraction. They save time and effort because they can automatically map dataflow graphs to the given architecture. The **CoreMap** design environment provides designers with the functionality to geometrically control the position of the modules on the FPGAs. This is helpful if partial reconfiguration has to be implemented. The main modules of the CoreMap are shown in figure 7.1.

7.1 The Graphical Editor

In the graphical editor of common CAD tools, components can be selected from a library, placed on a surface, and connected together to build complex modules. After this step, technology mapping, placement and routing are then required to provide a final implementation of the design. Generally, the design is flattened before mapping and placement. This step can alter the placement of a core at a contiguous location in the FPGA, thus making it difficult to locate modules on the FPGA surface. Moreover, all the steps required from synthesis to implementation are NP-complete problems which require long optimization times which increase compilation time. The CoreMap graphical editor follows the WYSIWYG (what you see is what you get) approach. The surface of the target FPGA is presented to the user as a grid. Each point on the grid represents a CLB in the FPGA (Fig 7.2). Operators like adders, multipliers, subtractors, multiply-accumulators (MAC), CORDIC (Coordinate Rotation Digital Computer) are pre-synthesized and kept in a library. The user can now select the pre-synthesized components from the library, place them on the FPGA

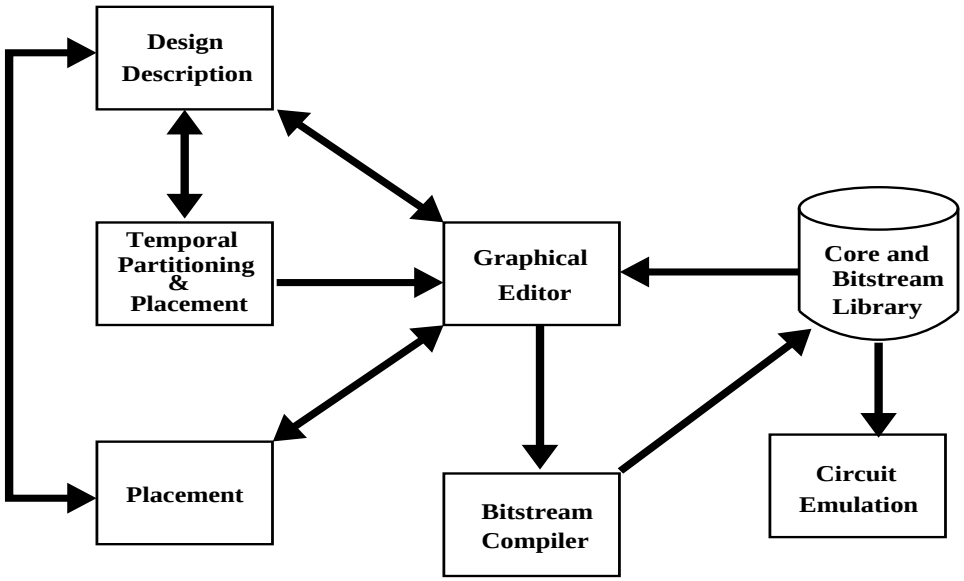


Figure 7.1: The CoreMap design flow

surface, and connect them together to build his design. The design is implemented on the FPGA and the components are placed at the locations chosen by the user. The resulting designs are stored either in the GML (Graph Modeling Language) or the XML (Extended Markup Language) format to allow their use in third party tools.

7.2 The Bitstream Compiler

The Bitstream compiler is the central module in the CoreMap. It takes a description of a design in GML or XML format and produces the bitstream needed to program the FPGA. A design description consists of the modules description, their position, their orientation and their interconnection. These are the parameters required to configure the pre-synthesized modules which are stored as skeletons in the library. The FPGA is programmed by setting the correct values for the function into the LUTs and connecting the inputs and outputs as specified in the boolean equation of the function to be implemented. The bitstream compiler uses the Xilinx JBits API [63] to generate the bitstream which will be used to program the FPGA.

7.3 Temporal Partitioning and Temporal Placement

The CoreMap features a temporal partitioning and temporal placement modules.

7.3.1 Temporal Partitioning

Temporal partitioning is computed by the methods described in chapter 5. The input is a dataflow graph in GML format. The user can choose to partition the DFG either with list-scheduling or with the spectral method or to let the system automatically choose the best method for temporal partitioning. In this case the system will decide which methods should be used on the basis of the connectivity of the dataflow graph. For each component, the partitioning procedure determines the partition the component belongs to as well as its position on the FPGA. For graphs computed with the spectral method, the definitive positions of component on the FPGA are determined by

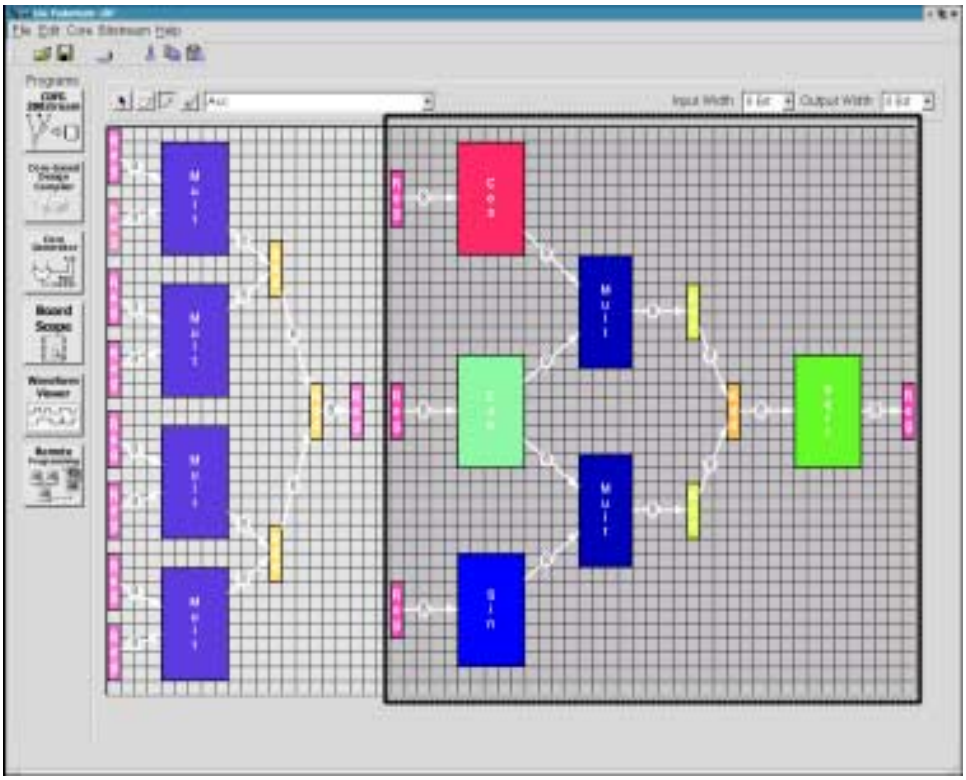


Figure 7.2: The CoreMap Graphical Design Environment

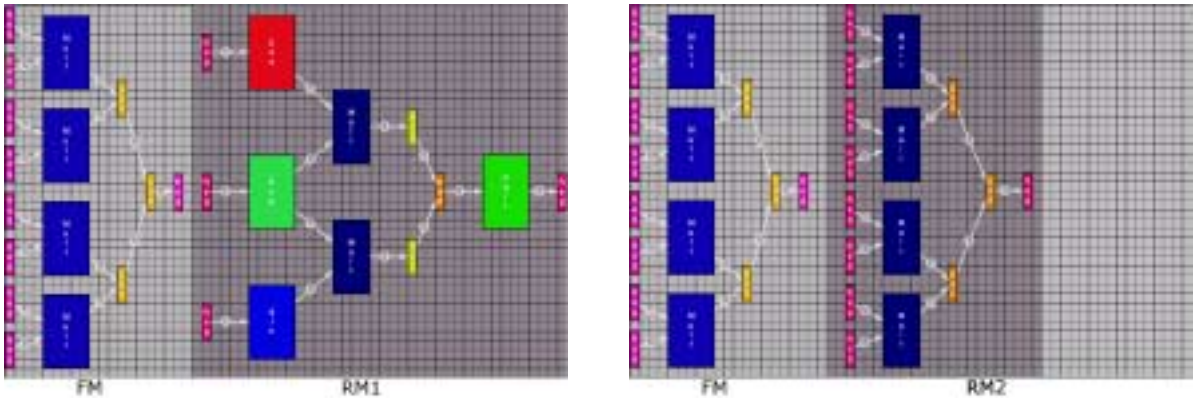
a procedure which scales the x and y-coordinates of the components by the values of their width and height. This is done because the spectral placement method does not take the bounding box of the component into account during the partitioning. If list-scheduling was used, a two dimensional spectral placement will be used to place the component of each partition on the FPGA surface. Once the definitive positions are computed, the JBits API is used to generate a java program which contains the instructions used to produce a bitstream for each partition. The bitstreams are then generated by compiling and running the java program.

7.3.2 Temporal Placement

The temporal placement can be applied only if a device supports partial reconfiguration. If this is the case, then a tool which implements the temporal placement should provide a way to implement partial reconfiguration. We will first explain how CoreMap implements partial reconfiguration.

Support for Partial Reconfiguration

For two consecutives configurations ζ_i and ζ_{i+1}, the system should be able to compute the set of packets needed to move from ζ_i to ζ_{i+1} without reconfiguring the entire device. This set of packets is computed on the basis of the difference $\zeta_{i+1} - \zeta_i$. This is done in the CoreMap by placing the set of clusters required in any configuration together. For each configuration, a bitstream is generated. The set of packets needed to partially reconfigure the device is then computed by a subtraction of the two consecutive bitstreams. The example in figure 7.3 illustrates this approach. In this example, a fix module (FM) for computing the matrix multiplication is placed on the FPGA together with a first reconfigurable module (RM1) for computing the Fourier summation in the first configuration. In the second configuration, the fix module is placed on the FPGA with a second reconfigurable



(a) Modules of the first bitstream

(b) Modules of the second bitstream

Figure 7.3: Implementation of partial reconfiguration in the CoreMap

module (RM2) which replaces the Fourier summation. By computing the difference between the two bitstreams the system is able to generate the partial bitstream needed to move from the first configuration to the next. In this case, only the marked column in the second configuration will be computed as a packet to be sent to the device for partial reconfiguration.

Temporal Placement

As in temporal partitioning, the input for temporal placement is a dataflow graph. The user can choose to cluster his design by one of the methods described in chapter 5 (level-based or spectral) or to let the system choose the appropriate method for the design. Additionally the user can provide the value of α and β as described in equation 5.17 and the number of slots (cluster blocks) on the FPGA. The default values are $\alpha = 2/3$ and $\beta = 1/3$. The system is actually able to compute the set of clusters to be "temporally" placed on the slots of the FPGA. The generation of the partial bitstream for the partial reconfiguration as explained earlier must be done by hand, but we are working on the automation of this step.

7.4 Circuit Emulation

CoreMap allows the user to test his design inside the FPGA. The input and output signals of a design are connected to register in the FPGA. These registers are mapped in the processor address space and can be accessed for writing or reading a signal (Fig 7.4). CoreMap provides a graphical user interface in which the input registers can be written and the output registers can be read. Register values can be displayed in different formats (binary, hexadecimal, decimal).

7.5 Multiprocessor Support

Additionally, CoreMap supports remote management of workstations equipped with an FPGA-board. Thus, it is possible to share expensive FPGA hardware within one working group. Bitstreams can be uploaded to a remote machine and tested with an interface developed for hardware-in-the-loop simulation. To establish a simple way of communication, CoreMap relies on TCP-

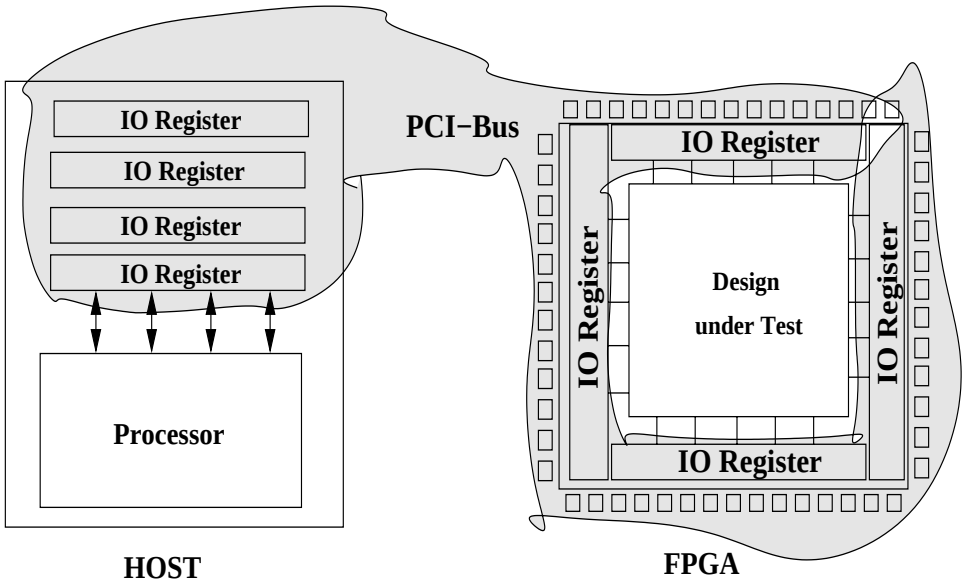


Figure 7.4: Design Emulation

sockets. A small server socket is running on each of the remote machines and listens for commands from a CoreMap client. To guarantee the correctness of commands and data packages, a CRC-checksum (cyclic redundancy check) is used. This helps to avoid data corruption during transmission and errors which can destroy the FPGA hardware.

7.6 Behind the CoreMap: The JBits API

CoreMap supports the Xilinx Virtex FPGAs [2] which are the most established and widely available Xilinx FPGAs. It uses the JBits library to generate the bitstream to program the FPGA. JBits is an application interface developed by Xilinx to allow the end user to set the content of the LUT and make connections inside the FPGA without the need for other CAD tools. JBits allows for very fine grained programming of FPGA. It has great potential for reconfigurable computing, since the problems with the current design methodologies (section 1.2) can partly be solved by full control of single bit and connection inside the FPGA.

The JBits API is constructed from a set of java classes, methods and tools which can be used to

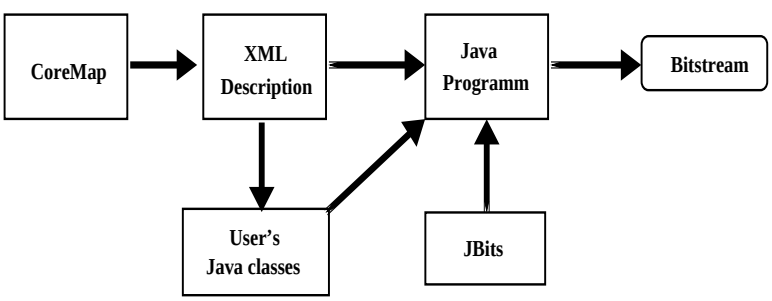


Figure 7.5: The CoreMap bitstream generation flow.

set the LUT-values as well as the interconnections. This is all that's required to implement our function in a FPGA. JBits provides functions to read back the content of a FPGA currently in use.

The content of a LUT can be set in using the set function:

$$set(row, col, SliceNumber\_Type, value) \tag{7.1}$$

which means that the content of the LUT *Type* (*Type* is either F or G (Fig 2.2)) in the slice *Number* (*number* can be 0 or 1) of the CLB in position *row* and *col* should be set to the value of the array *value*. The Virtex LUTs have four inputs and 16 entries representing the 16 possible values a 4-inputs function can take. A connection is defined using the function connect:

$$connect(outpin, inpin) \tag{7.2}$$

This function uses the JBits routing capabilities to connect the pin *outpin* to the pin *inpin* anywhere inside the FPGA. *outpin* and *inpin* should be one of the CLB terminals. Connecting the output of a LUT to a CLB terminal can be done with the function.

$$set(row, col, terminal\_control, LUT\_output) \tag{7.3}$$

where *terminal_control* set the correct terminal to be connected to the output *LUT_output* of the corresponding LUT. JBits provides hundreds of predefine cores like adders, subtractors, multipliers, CORDIC Processor, encoder and decoders, network modules, etc... which can directly be used in designs. They can also be combined to generate more complex cores.
As stated in section 1.2, implementing a function in FPGA with current design tools is a difficult task which can take days, weeks, or even months. Apart from the learning curve imposed on designers to master programming tools, incompatibility arising between the various tools needed from synthesis through programming remains a major problem. Experience is required to solve the problems arising during the design flow. In light of this, people not familiar with HDL, like mechanical engineers and computer scientists, must spend too much time learning HDL and complex CAD tools, thus neglecting the tasks that make best use their highest competencies. CoreMap [20] provides a simple design environment (Fig 7.1) which can help beginners, software engineers, and people not familiar with HDL to program FPGAs with less effort.

Chapter 8

Conclusion and Outlook

8.1 Summary

In this thesis we have investigated the synthesis of dataflow graphs for reconfigurable systems. After a brief overview on the evolution of reconfigurable devices in the last decade in chapter 1, we identified the challenges for the future of reconfigurable systems and defined the aims of our work. In chapter 2 we gave a brief introduction to the target reconfigurable devices (FPGAs) and defined our target platform in which the processor and the FPGA compute in parallel and exchange computing data as well as reconfiguration data. We formally defined the minimization of the amount of data exchange between the FPGA and the processor as the optimization goal for the problems to be solved in this thesis, i.e. the temporal partitioning and temporal placement that we stated in chapter 3. Since temporal placement can be done only if a device is partially reconfigurable, a temporal method has to be developed with the reconfiguration capability of the target device in mind. In our case, the target devices are the Xilinx Virtex FPGAs for which partial reconfiguration can be done only by exchanging complete columns. By exploiting this characteristic of the Virtex , we proved that grouping the components of a dataflow graph in clusters is very important in temporal placement. Our survey of previous work, reviewed in chapter 4, revealed that no other authors have stated the problem in this way. We showed that none of the earlier approaches was well suited to our needs. This motivated us to develop a fresh approach for solving the temporal partitioning and temporal placement problems. We made our contribution in chapter 5 through the development of various algorithms to solve the temporal partitioning and temporal placement problems. For each of the problems, we developed two methods, and provided criteria for selecting one method over the other.

To solve the temporal partitioning problem, we provided two methods. The first one is an enhancement of the well known *list scheduling* method for area optimization by the use of the so called configuration switching which maintain two configurations in the device and uses a condition signal to switch from one to the other. The second and completely novel method uses a three dimensional spectral placement to position the modules in a three dimensional vector space. The partitioning is done by picking component along the time-axis in increasing order of the time-coordinates. This approach is incremental and does not guaranty a cycle free configuration graph. Therefore, we used the Kernighan-Lin [93, 53, 98] algorithm to move the nodes of the graph from one partition to the other and insure a unidirectional partition. We introduce a modified gain function adapted to our need. The goal is not a minimum cut size like it is the case in many KL-FM algorithms, but a partition with all the edges converging in the same direction.

For temporal placement, the device is first divided in to slots. A slot is a set of consecutive columns. The dataflow graph is then divided in to clusters to be placed on the slots. A first-fit approach is

used to place the clusters on the slot at different times. For the computation of clusters, we provide two methods. The first one is a level-based clustering approach similar to list-scheduling partitioning, and the second is a spectral clustering method. With the spectral clustering method, the components of the given dataflow graph are first mapped into a 2-D vector space. A cluster growth approach is then used to build the clusters.
To evaluate the different methods, we defined two main criteria. The first one, the connectivity, is used to evaluate how strongly connected the components are in partitions or in clusters. If the components of a partition are strongly connected, then most of the edges will be inside the partitions, and therefore the communication among the clusters or partitions will be minimized. If this is not the case, then the edges will be outside of the clusters and therefore the communication among the clusters will be high. The second evaluation criterion is the wasted resource of a partitioning. This is defined as the surface occupied by the inactive components inside the clusters. For a given cluster, the greater the difference between the run-time of the cluster (this is the largest run-time over all its components) and the run-time of a given component, the greater the component wasted resource. Therefore, the need to place both components connected in the same cluster and components with approximately the same run-time is high. The construction of the level-based method made in chapter 5 ensures that components with approximately the same run-time are placed in the same cluster. The level-base method is likely to provide clustering with fewer wasted resources. To ensure that the spectral performs as well, we provided a formula (equation 6.1) to capture the time factor in the connection matrix used for the spectral placement.
Based on a benchmark of randomly generated graphs, we provided an evaluation of our methods in chapter 6. the results support the different assumptions we made in chapter 5 and 6 on the connectivity and the level of wasted resource of the developed method.
The methods developed are integrated in a simple and useful design environment, called CoreMap, that we presented in chapter 7.

8.2 Outlook

So far we have dealt with the synthesis of directed acyclic graphs. But algorithms are not restricted to dataflow graphs. Control mechanisms should also be considered. The extension of this work can proceed by the temporal partitioning of control dataflow graphs containing bidirectional edges. As we explained in chapter 3, a multi-FPGA temporal partitioning should be targeted in this case. If the generated configuration graph contains bidirectional edges or cycles, then all the nodes of the configuration graph in the path defined by the cycle should be placed at the same time on the set of available FPGAs. Using this approach, we can divide any kind of circuit for temporal partitioning. This allows us to provide a fast and cheap emulation solution by partitioning very large designs on to a small set of FPGAs. We believe that FPGAs will remain one of the preferred solutions for circuit emulation in the future. This suggests that temporal partitioning and temporal placement will continue to be hot topics going forward. As the trend towards falling FPGA prices and rising capacities continue, and new tools are delivered to implement partial reconfiguration, FPGAs will become increasingly attractive for mass production. Today, a partially configurable Xilinx FPGA can be purchased for less than 10 USD. This opens many possibilities in mechatronics, where adaptive and intelligent controllers can be exchanged without interrupting the control system, and in consumer electronics, where it drives single chip functional versatility across game, music and video product lines. Partial bitstreams can be generated for each module required. Our temporal placement algorithm is useful for clustering and the temporal placement of the clusters at different times on the FPGA.

Bibliography

[1] *RC-1000 Hardware, Software and Funktion reference Manual*, 1999.

[2] *Xilinx Virtex 2.5V Field programmable Gate Arrays: Preliminary product specification*, 1999.

[3] *JBits documentation*, 2001.

[4] *Xilinx Data Source CD-ROM*, 2002.

[5] A. M. S. Adario and S. Bampi. Reconfigurable computing: Viable applications and trends. In *IFIP TC10 WG10.5 10 Int. Conf. on Very Large Scale Integration(VLSI'99)*, pages 583–594, Lisboa, Portugal, 1999. IFIP.

[6] C. Alpert and A. Kahng. Geometric embeddings for faster and better multi-way netlist partitioning, 1993.

[7] C. Alpert and A. Kahng. Multi-way system partitioning via geometric embeddings, 1994.

[8] C. J. Alpert and A. B. Kahng. Multi-way partitioning via spacefilling curves and dynamic programming. In *Design Automation Conference*, pages 652–657, 1994.

[9] C. J. Alpert and S.-Z. Yao. Spectral partitioning: The more eigenvectors, the better. In *Design Automation Conference*, pages 195–200, 1995.

[10] P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3):11–18, 1993.

[11] R. Baeza-Tates. *Handbook of Algorithms and Data Structures*. Addison-Wessley, 1991.

[12] E. R. Barnes. An algorithm for partitioning the nodes of a graph. *SIAM Journal on Algorithms and Discrete Methods*, pages 541–550, December 1982.

[13] E. R. Barnes, A. Vannelliand, and J. Q. Walker. A new heuristic for partitioning the nodes of a graph. *SIAM Journal on Algorithms and Discrete Methods*, pages 229–305, August 1988.

[14] M. Berry, T. Do, G. O'Brien, V. Krishna, and S. Varadhan. Using linear algebra for information retrieval. *J. Soc. Indust. Appl. Math.*, 37(4):573–595, 1995.

[15] M. Berry, T. Do, G. O'Brien, V. Krishna, and S. Varadhan. *SVDPACK(Version 1.0) User's Guide*, 1996.

[16] C. Bobda. IP based synthesis of reconfigurables systems. In *Tenth ACM International Symposium on Field Programmable Gate Arrays(FPGA 02)*, page 248, Monterey, California, 2002. ACM/SIGDA.

[17] C. Bobda. Temporal partitioning and sequencing of dataflow graphs on reconfigurable systems. In *International IFIP TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002)*, pages 185–194, Montreal, Canada, 2002. IFIP.

[18] C. Bobda and T. Lehmann. Efficient building of word recongnizer in fpgas for term-document matrices construction. In R. Hartenstein and H. Grünbacher, editors, *Field Programmable Logic and Aplications FPL 2000*, pages 759–768, Villach, Austria, 2000. Springer.

[19] C. Bobda and N. Steenbock. Singular value decomposition on distributed reconfigurable systems. In *12th IEEE International Workshop On Rapid System Prototyping(RSP'01), Monterey California*. IEEE Computer Society, 2001.

[20] C. Bobda and N. Steenbock. A rapid prototyping environment for distributed reconfigurable systems. In *13th IEEE International Workshop On Rapid System Prototyping(RSP'02), Darmstadt Germany*. IEEE Computer Society, 2002.

[21] R. P. Brent. Parallel algorithms in linear algebra. In *Proc. Second NEC Research Symposium*, Tsukuba, Japan, August 1991.

[22] R. P. Brent and F. T. Luk. The solution of singular-value and eigen-value problems on multiprocessor arrays. *SIAM J. Sci. Stat. Comput.*, 6(1):69–84, 1985.

[23] K. Buchenrieder. *HW/SW Co-Design An annotated Bibilography*. IT Press, Chicago, 19994.

[24] D. A. Buel, J. M. Arnold, and W. J. Kleinfelder. *Splash 2 FPGAs in a Custom Computing Machine*. IEEE Computer Society Press, Los Alamitos, California, 1996.

[25] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 2000.

[26] T. J. Callahan, P. Chong, A. Dehon, and J. Wawrzynek. Fast module mapping and placement for datapaths in fpgas. In *International Symposium on Field Programmable Gate Arrays(FPGA 98)*, pages 123 – 132, Monterey, California, 1998. ACM/SIGDA.

[27] J. M. P. Cardoso and H. C. Neto. An enhance static-list scheduling algorithm for temporal partitioning onto rpus. In *IFIP TC10 WG10.5 10 Int. Conf. on Very Large Scale Integration(VLSI'99)*, pages 485 – 496, Lisboa, Portugal, 1999. IFIP.

[28] W. S. Carter, K. Duong, R. H. Freeman, H. C. Hsieh, J. Y. E. Mahoney, L. T. Ngo, and S. L. Sze. A user programmable gate array. In *IEEE 1986 Custom Integrated Circuits Conference*, pages 233–235, 1986.

[29] P. Chan, M. Schlag, and J. Zien. SPECTRAL-BASED MULTI-WAY FPGA PARTITIONING. Technical Report UCSC-CRL-94-44, 1994.

[30] D. Chang and M. Marek-Sadowska. Partitioning sequential circuits on dynamicaly reconfigurable fpgas. In *International Symposium on Field Programmable Gate Arrays(FPGA 98)*, pages 161 – 167, Monterey, California, 1998. ACM/SIGDA.

[31] C. Cheng and E. Kuh. Module placement based on resistive network optimization, 1984.

[32] W. Cockshott and P. Foulk. A low-cost text retrieval machine. *IEEE PROCEEDINGS*, 136(4):271–276, July 1989.

[33] K. Compton, J. Cooley, S. Knol, and S. Hauck. Configuration relocation and defragmentation for fpgas, 2000.

[34] K. Compton and S. Hauck. Configurable computing: A survey of systems and software, 1999.

[35] A. K. S. Deepali Deshpande and A. Tyagi. Configuration caching vs caching for striped fpgas. In *International Symposium on Field Programmable Gate Arrays(FPGA 98)*, pages 206 – 214, Monterey, California, 1999. ACM/SIGDA.

[36] S. Deerwester, S. Dumai, G. Furnas, T. Landauer, and R. Harshmann. Indexing by latent semantic analysis. *Journal of American Society for Information Science*, 41(6):391–407, 1990.

[37] O. Diessel and H. ElGindy. Run-time compaction of fpga designs. In *Field-Programmable Logic and Applications*, London, UK, September 1997. Springer.

[38] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt. Dynamic scheduling of tasks on partially reconfigurable fpgas, 2000.

[39] O. Diessel and H. A. ElGindy. Partial FPGA rearrangement by local repacking (abstract). In *FPGA*, page 259, 1998.

[40] W. E. Donath. Hierarchical placement method. *IBM Technical disclosure Bulletin*, 17(10):3121–3125, 1975.

[41] W. E. Donath and A. J. Hoffman. Algorithms for partitioning of graphs and computer logic based on eigenvectors of connection matrices. *IBM Technical disclosure Bulletin*, 15(3):938–944, 1972.

[42] E. Ed. Datapath-oriented fpga mapping and placement for configurable computing.

[43] M. Eisenring and M. Platzner. Optimazition of run-time reconfigurable embedded systems. In R. Hartenstein and H. Grünbacher, editors, *Field Programmable Logic and Aplications FPL 2000*, pages 565–574, Villach, Austria, 2000. Springer.

[44] Ejnioui and N. Ranganathan. Circuit scheduling on time-multiplexed fpgas.

[45] J. G. Eldredge and B. L. Hutchings. Run-time reconfiguration: A method for enhancing the functional density of SRAM-based FPGAs. *Journal of VLSI Signal Processing*, 12:67–86, 1996.

[46] H. ElGindy, M. Middendorf, H. Schmeck, , and B. Schmidt. Task rearrangement on partially reconfigurable fpgas with restricted buffer. In R. Hartenstein and H. Grünbacher, editors, *Field Programmable Logic and Aplications FPL 2000*, pages 656–664, Villach, Austria, 2000. Springer.

[47] F. Embedde Systems Design Group. *Spyder-Virtex-X2 User's manual*, 1999.

[48] J. M. Emmert and D. K. Ahatia. Tabu search: Ultra-fast placement for fpgas. In P. Lysaght and J. Irvine, editors, *Field Programmable Logic and Aplications FPL 1999*, pages 81–90, Glasgow, UK, 1999. Springer.

[49] G. Estrin, B. Bussell, R. Turn, and J. Bibb. Parallel processing in a restructurable computer system. *IEEE Transactions on Electronic Computers*, 12(5):747–755, 1963.

[50] G. Estrin and R. Turn. Automatic assignment of computations in a variable structure computer system. *IEEE Transactions on Electronic Computers*, 12(5):755–773, 1963.

[51] G. Estrin and C. R. Viswanathan. Organisation of a ”fixed-plus-variable” structure computer for computation of eigenvalues and eigenvectors of real symetric matrices. *Journal of the ACM*, 9:41–60, 1962.

[52] S. P. Fekete, E. Khler, and J. Teich. Optimal fpga module placement with temporal precedence constraints. Technical Report 696.2000, Technische Universität Berlin, 2000.

[53] C. M. Fiduccia and R. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, pages 175–181, 1982.

[54] L. Ford and D. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[55] G. E. Forsythe and P. Henrici. The cyclic jacobi method for computing the principal values of a complex matrix. *Trans. Amer. Math. Soc.*, 94:1–23, 1960.

[56] P. Foulk. Data-folding in SRAM configurable FPGA. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 163–171. IEEE, 1993.

[57] C. H. Gebotys. An optimal methodology of synthesis of dsp multichip architectures. *Journal of VLSI Signal Processing*, 11:9–19, 1995.

[58] G. H. Golub and C. F. V. Loan. *Matrix Computations*. North Oxford Academic Publisching, 1983.

[59] Guattery and Miller. On the performance of spectral graph partitioning methods. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1995.

[60] S. Guattery and G. L. Miller. On the quality of spectral separators. *SIAM Journal on Matrix Analysis and Applications*, 19(3):701–719, 1998.

[61] S. Guccione. *Programming Fine-Grained Reconfigurable Architechture*. PhD thesis, The University of Texas at Austin, May 1995.

[62] S. Guccione and D. Levi. The advantages of run-time reconfiguration. In *Proc. SPIE 3844 FPGAs for Computing and Applications*, pages 87–92. John Chewel et al, Eds, Sept. 1999.

[63] S. Guccione, D. Levi, and P. Sundararajan. Jbits: A java-based interface for reconfigurable computing, 1999.

[64] B. Gunther and G. Milne. Accessing document relevance with run-time reconfigurable machines. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 9–16, Napa California, 1996. IEEE.

[65] B. Gunther and G. Milne. Hardware-based solution for message filtering. Technical report, school of computer and information science, 1996.

[66] A. Haase, R. Siegmund, C. Kretzschmar, D. Mueller, J. Schneider, M. Boden, and M. Langer. Design of a reed solomon decoder using partial dynamic reconfiguration of xilinx virtex fpgas - a case study. In *Design Automation and Test in Europe, Designers Forum*, March 2001.

[67] J. Hadley and B. Hutchings. Design methodologies for partially reconfigured systems. In P. Athanas and K. L. Pocek, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 78–84, Los Alamitos, CA, 1995. IEEE Computer Society Press.

[68] J. D. Hadley and B. L. Hutchings. Designing a partially reconfigured system. In J. Schewel, editor, *Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, Proc. SPIE 2607*, pages 210–220, Bellingham, WA, 1995. SPIE – The International Society for Optical Engineering.

[69] K. Hall. An r-dimensional quadratic dimensional quadratic placement algorithm. *Journal of Management Science*, 17(3):219–229, 1970.

[70] E. R. Hansen. On cyclic jacobi methods. *J. Soc. Indust. Appl. Math.*, 11(2):448–459, 1963.

[71] W. Hardt. *HW/SW-Codesign auf Basis von C-Programmen unter Performanz-Gesichtpunkten*. PhD thesis, Universität GH Paderborn, Germany, 1996.

[72] W. Hardt and R. Camposano. Kriterien zur hardware/software-partitionierung beim systementwurf. In *Proc. of the Workshop des Sonderforschungsbereiches 358 der TU Dresden und der GI/ITG*. GI/ITG, 1993.

[73] W. Hardt and R. Camposano. Specification analysis for hw/sw partitioning. Technical Report SFB 358 - B2 - 5/94, University of Paderborn, Technical University of Dresden, 1994.

[74] W. Hardt and R. Camposano. Trade-offs in hw/sw-codesign. In *Proc. of the 3rd ACM Workshop on Hardware/Software Codesign.*, Cambridge, MA, Oktober 1993. GI/ITG.

[75] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. Using the KressArray for reconfigurable computing. In J. Schewel, editor, *Configurable Computing: Technology and Applications, Proc. SPIE 3526*, pages 150–161, Bellingham, WA, 1998. SPIE – The International Society for Optical Engineering.

[76] R. Hartenstein, A. Hirschbiel, and M.Weber. Xputers - an open family of non von neumann architectures. In *11th ITG/GI Conference on Architektur von Rechensystemen*. VDE-Verlag, 1990.

[77] R. W. Hartenstein, M. Riedmuller, K. Schmitt, and M. Weber. A novel asic design approach based on a new machine paradigm, 1996.

[78] J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In K. L. Pocek and J. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[79] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16(2):452–469, 1995.

[80] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers Inc, 1990.

[81] M. R. Hestenes. Inversion of matrices by biorthogonalization and related results. *J. Soc. Indust. Appl. Math.*, 6(1):51–90, 1958.

[82] E. L. Horta, J. W. Lockwood, D. E. Taylor, and D. Parlour. Dynamic hardware plugins (dhps) in an fpga with partial run-time reconfiguration (rtr). In *Tenth ACM International Symposium on Field Programmable Gate Arrays(FPGA 02)*, page 250, Monterey, California, 2002. ACM/SIGDA.

[83] S. Iman, M. Pedram, C. Fabian, and J. Cong. Finding uni-directional cuts based on physical partitioning and logic restructuring. In *Fourth International Workshop on Physical Design*. IEEE, 1993.

[84] C. inc. *DK1 Reference Manual*, 2000.

[85] C. inc. *Handel-C Reference Manual*, 2000.

[86] M. G. Jan. Early experience with a hybrid processor: K-means clustering.

[87] X. jie Zhang, Kam-wing, and W. Luk. A combined approach to high-level synthesis for dynamicaly reconfigurable systems. In R. Hartenstein and H. Grünbacher, editors, *Field Programmable Logic and Aplications FPL 2000*, pages 361–370, Villach, Austria, 2000. Springer.

[88] H. Kalte, M. Pormann, and U. Rueckert. Rapid prototyping system f”ur dynamisch rekonfigurierbarer hardware struckturen. In *AES 2000*, pages 149–157, 2000.

[89] R. Kamdem and P. Njiwoua. Galois lattice approach to hardware/software partitioning. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 99)*, 1999.

[90] M. Kaul and R. Vemuri. Optimal temporal partitioning and synthesis for reconfigurable architectures, 1998.

[91] M. Kaul, R. Vemuri, S. Govindarajan, and I. Ouaiss. An automated temporal partitioning tool for a class of dsp applications, 1998.

[92] T. Kean. *Configurable Logic: A Dynamically Programmable Cellular Architecture and its VLSI Implementation*. PhD thesis, University of Edimburgh, 1989.

[93] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.

[94] H. Krupnova and G. Saucier. Hierachical interactive approach to partition large designs into fpgas. In P. Lysaght and J. Irvine, editors, *Field Programmable Logic and Aplications FPL 1999*, pages 102–110, Glasgow, UK, 1999. Springer.

[95] K. Kucukcakar and A. Parker. Chop: A constraint-driven-system-level partitioner. In *Design Automation Conference*, pages 514–519, 1991.

[96] P. Kurup and T. Abbasi. *Logic Synthesis Using Synopsys*. Kluwer Akademic publisher, 1997.

[97] T. Lehmann and A. Schreckenberg. Case study of integration of reconfigurabel logic as a coprocessor into a sci-cluster under rt-linux. In *Field-Programmable Logic and Applications*, Belfast, Northern Irland, August 2001. Springer.

[98] T. Lengauer. *Combinatorial Algorithm for Integrated Circuit Layout*. Teubner Stuttgart, 1990.

[99] H. Liu and D. F. Wong. Network flow-based circuit partitioning for time-multiplexed FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 497–504, 1998.

[100] H. Liu and D. F. Wong. Circuit partitioning for dynamicaly reconfigurable fpgas. In *International Symposium on Field Programmable Gate Arrays(FPGA 98)*, pages 187 – 194, Monterey, California, 1999. ACM/SIGDA.

[101] S.-M. Ludwig. *Hades-Fast Hardware Synthesis Tools and a Reconfigurable Coprocessor*. PhD thesis, Swiss Federal Institute of Technologie, Zürich, 1997.

[102] F. T. Luk and H. Park. On parallel jacobi orderings. *SIAM J. Sci. Stat. Comput.*, 10(1):18–26, 1989.

[103] W. Mangione-Smith and B. Hutchings. Configurable computing: The road ahead. In R. Hartenstein and V. Prasanna, editors, *Reconfigurable Architectures: High Performance by Configware*, pages 81–96, Chicago, 1997. IT Press.

[104] J. McMillan and S. A. Guccione. Partial run-time reconfiguration using jrtr. In R. W. Hartenstein and H. Grunbacher, editors, *Field Programmable Logic and Aplications FPL 2000*, pages 352–360, Villach, Austria, 2000. Springer.

[105] L. Minzer. Programmable silicon for embedde signal processing. *Embedded Systems Programming*, pages 110–133, March 2000.

[106] U. o. P. MLaP-Mechatronics Laboratory. *RABBIT FPGA Module user's manual*, 1999.

[107] L. Moll, J. Vuillemin, and P. Boucard. High energy physics on DECPeRLe-1 programmable active memory. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 47–52, Monterey, CA, 1995.

[108] I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri. An integrated partitioning and synthesis system for dynamically reconfigurable multi-FPGA architectures. In *IPPS/SPDP Workshops*, pages 31–36, 1998.

[109] Y. Pan and M. Hamdi. Singular value decomposition on processors arrays with a pipelined bus system. *Journal of Network and Computer Applications*, 19:235–248, 1996.

[110] A. Pandey and R. Vemuri. Combined temporal partitioning and scheduling for reconfigurable architectures. In J. Schewel, P. M. Athanas, S. A. Guccione, S. Ludwig, and J. T. McHenry, editors, *Reconfigurable Technology: FPGAs for Computing and Applications, Proc. SPIE 3844*, pages 93–103, Bellingham, WA, 1999. SPIE – The International Society for Optical Engineering.

[111] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of asic's. *IEEE Transactions on CAD*, 6(8):661–679, 1989.

[112] K. M. G. Purna and D. Bhatia. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *IEEE Transactions on Computers*, 48(6):579–590, 1999.

[113] A. H. R. Genevriere. Pmoss - a modular synthesis and hw/sw-codesign system. Technical Report SFB 358 - B2 - 2/94, University of Paderborn, 1994.

[114] F. J. Rammig. A concept for the editing of hardware resulting in an automatic hardware-editor. In *Proceedings of 14th Design Automation Conference*, pages 187–193, New Orleans, 1977.

[115] H. Rustishauser. The jacobi method for real symetric matrices. *Handbook for Automatic Computation*, Vol 2 (linear Algebra):202–211, 1971.

[116] G. Salton. *The SMART Retrieval System*. Prentice Hall,Inc, 1971.

[117] S. M. Scalera and J. R. Vzquez. The design and implementation of a context switching fpga. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 78–85, Napa Valey, CA, April 1998. IEEE Computer Society Press.

[118] N. Shirazi, A. Walters, and P. Athanas. Quantitative analysis of floating point arithmetic on fpga based custom computing machines. In *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, California, 1995. IEEE.

[119] C. Siemers. Rechnerfabrik: Ansaetze fuer etrem parallele prozessoren. *C't*, 15:170–179, 2001.

[120] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2:135–148, 1991.

[121] J. Teich, S. P. Fekete, and J. Schepers. Optimizing dynamic hardware reconfiguration. Technical Report 97.228, Angewante Mathematic Und Informatik Universität zu Köln, 1998.

[122] G. R. G. S. J. Thomas. An optimal parallel jacobi-like solution method for the singular value decomposition. In *Proc. Int. Conf. on Parallel Processing*, January 1988.

[123] S. Trimberger. Scheduling designs into a time-multiplexed fpga. In *International Symposium on Field Programmable Gate Arrays(FPGA 98)*, pages 153 – 160, Monterey, California, 1998. ACM/SIGDA.

[124] M. Vasilko. Dynasty: A temporal floorplanning based cad framework for dynamicaly reconfigurable logic systems. In P. Lysaght and J. Irvine, editors, *Field Programmable Logic and Aplications FPL 1999*, pages 124–133, Glasgow, UK, 1999. Springer.

[125] M. Vasilko and G. Benyon-Tinker. Automatic temporal floorplanning with guaranteed solution feasibility. In R. Hartenstein and H. Grünbacher, editors, *Field Programmable Logic and Aplications FPL 2000*, pages 656–664, Villach, Austria, 2000. Springer.

[126] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, 1996.

[127] J. H. Wilkinson. *The algebraic eigenvalue problem*. Oxford University Press, 1965.

[128] M. Wirthlin and B. Hutchings. A dynamic instruction set computer. In P. Athanas and K. L. Pocek, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 99–107, Los Alamitos, CA, 1995. IEEE Computer Society Press.

[129] A. C. H. Wu and D. D. Gajski. Partitioning algorithms for layout synthesis from register-transfer netlists. In *International Conference on Computer-Aided Design*, pages 144–147, 1990.

[130] H. Yang and D. F. Wong. Efficient network flow based min-cut balanced partitioning. In *International Conference on Computer-Aided Design*, 1994.